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

Gerwin Klein Tobias Nipkow

January 31, 2023

Contents

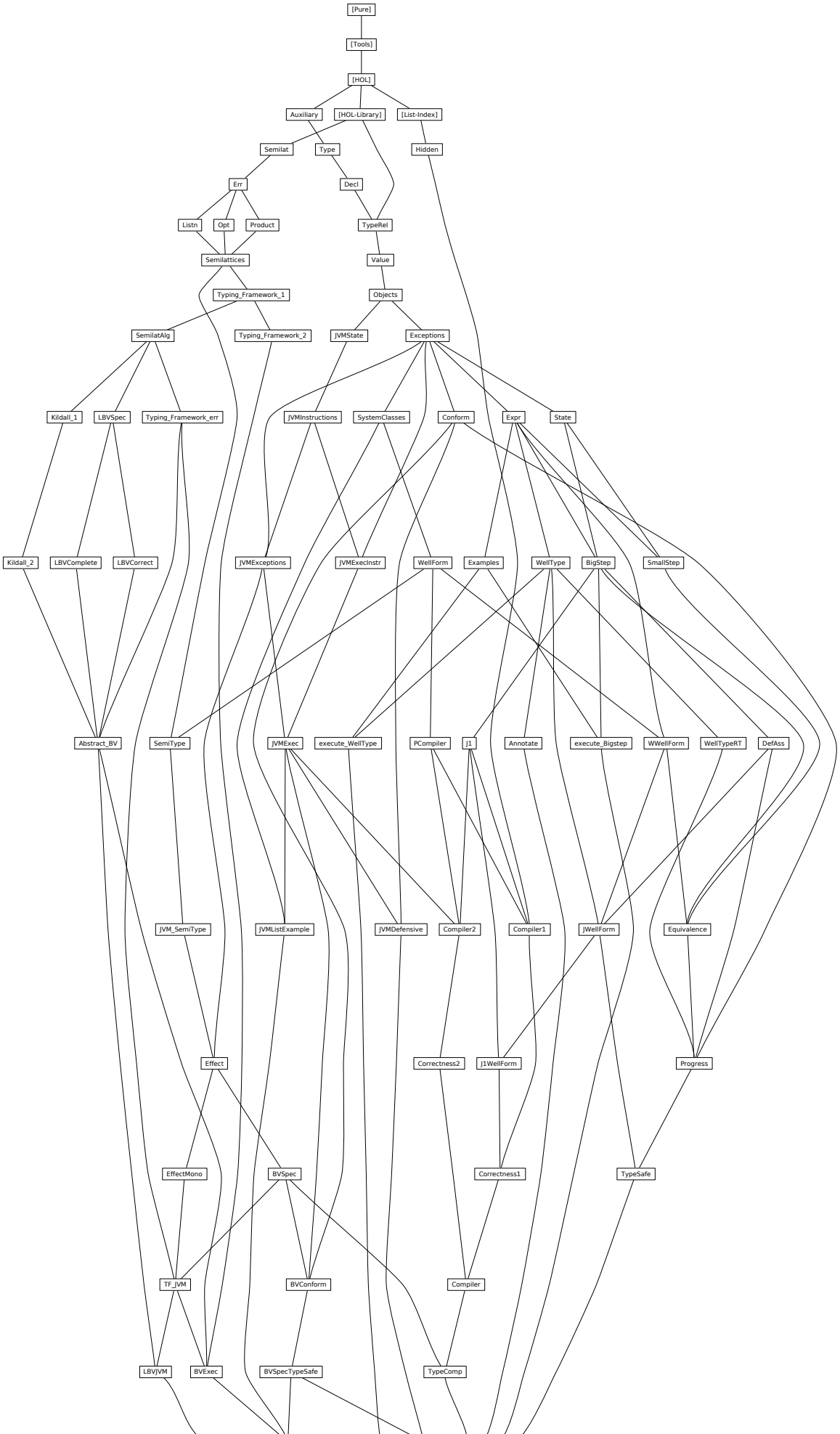
Chapter 1

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, 2].

1.1 Theory Dependencies

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


[Pure]
[Tools]
[HOL]
Auxiliary
[HOL-Library]
[List-Index]
Semilat
Type
Hidden
Err
Decl
Listn
Opt
Product
TypeRel
Semilattices
Value
Typing_Framework_1
Objects
SemilatAlg
Typing_Framework_2
JVMState
Exceptions
Kildall_1
LBVSpec
Typing_Framework_err
JVMInstructions
SystemClasses
Conform
Expr
State
Kildall_2
LBVComplete
LBVCorrect
JVMExceptions
JVMExecInstr
WellForm
Examples
WellType
BigStep
SmallStep
Abstract_BV
SemiType
JVMExec
execute_WellType
PCompiler
J1
Annotate
execute_Bigstep
WWellForm
WellTypeRT
DefAss
JVM_SemiType
JVMListExample
JVMDefensive
Compiler2
Compiler1
JWellForm
Equivalence
Effect
Correctness2
J1WellForm
Progress
EffectMono
BVSpec
Correctness1
TypeSafe
TF_JVM
BVConform
Compiler
LBVJVM
BVExec
BVSpecTypeSafe
TypeComp

Chapter 2

Jinja Source Language

2.1 Auxiliary Definitions

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

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

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

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

2.1.1 *distinct-fst*

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

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

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

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

2.1.2 Using *list-all2* for relations

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

Convenience lemmas

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

lemma *rel-list-all2-Cons1*:
list-all2 (*fun-of S*) (*x*#*xs*) *ys* =
(∃ *z zs*. *ys* = *z*#*zs* ∧ (*x*,*z*) ∈ *S* ∧ *list-all2* (*fun-of S*) *xs zs*)

lemma *rel-list-all2-Cons2*:
list-all2 (*fun-of S*) *xs* (*y*#*ys*) =
(∃ *z zs*. *xs* = *z*#*zs* ∧ (*z*,*y*) ∈ *S* ∧ *list-all2* (*fun-of S*) *zs ys*)

lemma *rel-list-all2-refl*:
(⋀*x*. (*x*,*x*) ∈ *S*) ⟹ *list-all2* (*fun-of S*) *xs xs*

lemma *rel-list-all2-antisym*:
⟦ (⋀*x y*. ⟦(*x*,*y*) ∈ *S*; (*y*,*x*) ∈ *T*⟧ ⟹ *x* = *y*);
list-all2 (*fun-of S*) *xs ys*; *list-all2* (*fun-of T*) *ys xs* ⟧ ⟹ *xs* = *ys*

lemma *rel-list-all2-trans*:
⟦ ⋀*a b c*. ⟦(*a*,*b*) ∈ *R*; (*b*,*c*) ∈ *S*⟧ ⟹ (*a*,*c*) ∈ *T*;
list-all2 (*fun-of R*) *as bs*; *list-all2* (*fun-of S*) *bs cs*⟧
⟹ *list-all2* (*fun-of T*) *as cs*

lemma *rel-list-all2-update-cong*:
⟦ *i*<*size xs*; *list-all2* (*fun-of S*) *xs ys*; (*x*,*y*) ∈ *S* ⟧
⟹ *list-all2* (*fun-of S*) (*xs*[*i*:=*x*]) (*ys*[*i*:=*y*])

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

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

end

2.2 Jinja types

theory *Type* **imports** *Auxiliary* **begin**

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

definition *Object* :: *cname*
where
Object ≡ ''*Object*''

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

— types
datatype *ty*
= *Void* — type of statements
| *Boolean*
| *Integer*

| *NT* — null type
| *Class cname* — class type

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

lemma [*iff*]: *is-refT NT*
lemma [*iff*]: *is-refT*(*Class C*)
lemma *refTE*:
⟦*is-refT T*; *T* = *NT* ⟹ *P*; ⋀*C*. *T* = *Class C* ⟹ *P* ⟧ ⟹ *P*
lemma *not-refTE*:
⟦ ¬*is-refT T*; *T* = *Void* ∨ *T* = *Boolean* ∨ *T* = *Integer* ⟹ *P* ⟧ ⟹ *P*
end

2.3 Class Declarations and Programs

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

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

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

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

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

definition *is-type* :: *'m prog* ⇒ *ty* ⇒ *bool*
where
is-type P T ≡
(*case T of Void* ⇒ *True* | *Boolean* ⇒ *True* | *Integer* ⇒ *True* | *NT* ⇒ *True*
| *Class C* ⇒ *is-class P C*)

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

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

end

2.4 Relations between Jinja Types

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

2.4.1 The subclass relations

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

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

lemma *subcls1D*: *P* ⊢ *C* ≺¹ *D* ⟹ *C* ≠ *Object* ∧ (∃*fs ms*. *class P C* = *Some* (*D*,*fs*,*ms*))
lemma [*iff*]: ¬ *P* ⊢ *Object* ≺¹ *C*
lemma [*iff*]: (*P* ⊢ *Object* ≼* *C*) = (*C* = *Object*)
lemma *subcls1-def2*:
subcls1 P =
(*SIGMA C*:{*C*. *is-class P C*}. {*D*. *C*≠*Object* ∧ *fst* (*the* (*class P C*))=*D*})
lemma *finite-subcls1*: *finite* (*subcls1 P*)

2.4.2 The subtype relations

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

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

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

lemma *Class-widen*: *P* ⊢ *Class C* ≤ *T* ⟹ ∃ *D*. *T* = *Class D*

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

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

2.4.3 Method lookup

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

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

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

definition *has-method* :: *'m prog* ⇒ *cname* ⇒ *mname* ⇒ *bool* (- ⊢ - *has* - [*51*,*0*,*51*] *50*)

where
$P \vdash C$ *has* $M \equiv \exists\, Ts\ T\ m\ D.\ P \vdash C$ *sees* $M{:}Ts \to T = m$ *in* D

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

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

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

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

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

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

2.4.4 Field lookup

inductive
Fields :: $['m$ *prog*, *cname*, $((vname \times cname) \times ty)$ *list*$] \Rightarrow$ *bool*
$(- \vdash - $ *has'-fields* $- [51,51,51]\ 50)$
for $P :: 'm$ *prog*
where
has-fields-rec:
$[\![$ *class* $P\ C = Some(D,fs,ms);\ C \neq Object;\ P \vdash D$ *has-fields* $FDTs;$
$FDTs' = map\ (\lambda(F,T).\ ((F,C),T))\ fs\ @\ FDTs\ ]\!]$
$\Longrightarrow P \vdash C$ *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* $pre@FDTs \wedge dom(map\text{-}of\ pre) \cap dom(map\text{-}of\ FDTs) = \{\}$

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

lemma *has-field-mono*:
assumes *has*: $P \vdash C\ has\ F{:}T\ in\ D$ **and** *sub*: $P \vdash C' \preceq^* C$
shows $P \vdash C'\ has\ F{:}T\ in\ D$

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

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

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

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

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

2.4.5 Functional lookup

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

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

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

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

2.4.6 Code generator setup

code-pred
(*modes*: $i \Rightarrow i \Rightarrow i \Rightarrow bool$, $i \Rightarrow i \Rightarrow o \Rightarrow bool$)
subcls1p
.
declare *subcls1-def* [*code-pred-def*]

code-pred
(*modes*: $i \Rightarrow i \times o \Rightarrow bool$, $i \Rightarrow i \times i \Rightarrow bool$)
[*inductify*]
subcls1
.
definition *subcls'* **where** *subcls'* $G = (subcls1p\ G)\widehat{\ }**$
code-pred
(*modes*: $i \Rightarrow i \Rightarrow i \Rightarrow bool$, $i \Rightarrow i \Rightarrow o \Rightarrow bool$)
[*inductify*]
subcls'
.

lemma *subcls-conv-subcls'* [*code-unfold*]:
$(subcls1\ G)\widehat{\ }* = \{(C,\ D).\ subcls'\ G\ C\ D\}$
by (*simp add*: *subcls'-def subcls1-def rtrancl-def*)

code-pred
(*modes*: $i \Rightarrow i \Rightarrow i \Rightarrow bool$)
widen
.

code-pred
(*modes*: $i \Rightarrow i \Rightarrow o \Rightarrow bool$)
Fields
.

lemma *has-field-code* [*code-pred-intro*]:
$[\![\ P \vdash C$ *has-fields* $FDTs$; *map-of* $FDTs\ (F,\ D) = \lfloor T \rfloor\ ]\!]$
$\Longrightarrow P \vdash C$ *has* $F{:}T$ *in* D
by(*auto simp add*: *has-field-def*)

code-pred
(*modes*: $i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow i \Rightarrow bool$, $i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow bool$)
has-field
by(*auto simp add*: *has-field-def*)

lemma *sees-field-code* [*code-pred-intro*]:
$[\![\ P \vdash C$ *has-fields* $FDTs$; *map-of* $(map\ (\lambda((F,\ D),\ T).\ (F,\ D,\ T))\ FDTs)\ F = \lfloor (D,\ T) \rfloor\ ]\!]$
$\Longrightarrow P \vdash C$ *sees* $F{:}T$ *in* D
by(*auto simp add*: *sees-field-def*)

code-pred
(*modes*: $i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow o \Rightarrow bool$, $i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow i \Rightarrow bool$,
$i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow bool$, $i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow bool$)
sees-field
by(*auto simp add*: *sees-field-def*)

code-pred
(*modes*: $i \Rightarrow i \Rightarrow o \Rightarrow bool$)
Methods
.

lemma *Method-code* [*code-pred-intro*]:
$[\![P \vdash C$ *sees-methods* Mm; $Mm\ M = \lfloor((Ts,\ T,\ m),\ D)\rfloor]\!]$
$\Longrightarrow P \vdash C$ *sees* M: $Ts \rightarrow T = m$ *in* D
by(*auto simp add*: *Method-def*)

code-pred
(*modes*: $i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow o \Rightarrow o \Rightarrow o \Rightarrow bool$,
$i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow bool$)
Method
by(*auto simp add*: *Method-def*)

lemma *eval-Method-i-i-i-o-o-o-o-conv*:
Predicate.eval (*Method-i-i-i-o-o-o-o P C M*) = (λ(Ts, T, m, D). $P \vdash C$ *sees* M:$Ts \rightarrow T$=m *in* D)
by(*auto intro*: *Method-i-i-i-o-o-o-oI elim*: *Method-i-i-i-o-o-o-oE intro*!: *ext*)

lemma *method-code* [*code*]:
method P C M =
Predicate.the (*Predicate.bind* (*Method-i-i-i-o-o-o-o P C M*) (λ(Ts, T, m, D). *Predicate.single* (D, Ts, T, m)))
apply (*rule sym*, *rule the-eqI*)
apply (*simp add*: *method-def eval-Method-i-i-i-o-o-o-o-conv*)
apply (*rule arg-cong* [**where** *f*=*The*])
apply (*auto simp add*: *Sup-fun-def Sup-bool-def fun-eq-iff*)
done

lemma *eval-Fields-conv*:
Predicate.eval (*Fields-i-i-o P C*) = (λ*FDTs*. $P \vdash C$ *has-fields FDTs*)
by(*auto intro*: *Fields-i-i-oI elim*: *Fields-i-i-oE intro*!: *ext*)

lemma *fields-code* [*code*]:
fields P C = *Predicate.the* (*Fields-i-i-o P C*)
by(*simp add*: *fields-def Predicate.the-def eval-Fields-conv*)

lemma *eval-sees-field-i-i-i-o-o-conv*:
Predicate.eval (*sees-field-i-i-i-o-o P C F*) = (λ(T, D). $P \vdash C$ *sees* F:T *in* D)
by(*auto intro*!: *ext intro*: *sees-field-i-i-i-o-oI elim*: *sees-field-i-i-i-o-oE*)

lemma *eval-sees-field-i-i-i-o-i-conv*:
Predicate.eval (*sees-field-i-i-i-o-i P C F D*) = (λT. $P \vdash C$ *sees* F:T *in* D)
by(*auto intro*!: *ext intro*: *sees-field-i-i-i-o-iI elim*: *sees-field-i-i-i-o-iE*)

lemma *field-code* [*code*]:
field P C F = *Predicate.the* (*Predicate.bind* (*sees-field-i-i-i-o-o P C F*) (λ(T, D). *Predicate.single*

(*D*, *T*)))
apply (*rule sym*, *rule the-eqI*)
apply (*simp add*: *field-def eval-sees-field-i-i-i-o-o-conv*)
apply (*rule arg-cong* [**where** *f*=*The*])
apply (*auto simp add*: *Sup-fun-def Sup-bool-def fun-eq-iff*)
done

2.5 Jinja Values

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

type-synonym *addr* = *nat*

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

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

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

primrec *default-val* :: *ty* ⇒ *val* — default value for all types **where**
  default-val Void = *Unit*
| *default-val Boolean* = *Bool False*
| *default-val Integer* = *Intg 0*
| *default-val NT* = *Null*
| *default-val* (*Class C*) = *Null*

end

2.6 Objects and the Heap

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

2.6.1 Objects

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

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

definition *init-fields* :: ((*vname* × *cname*) × *ty*) *list* ⇒ *fields*
where

init-fields $\equiv$ *map-of* $\circ$ *map* $(\lambda(F,T).\ (F,$*default-val* $T))$

— a new, blank object with default values in all fields:
definition *blank* :: *'m prog* $\Rightarrow$ *cname* $\Rightarrow$ *obj*
where
blank P C $\equiv$ $(C,$*init-fields* (*fields P C*)$)$

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

2.6.2 Heap

type-synonym *heap* = *addr* $\rightharpoonup$ *obj*

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

definition *new-Addr* :: *heap* $\Rightarrow$ *addr option*
where
new-Addr h $\equiv$ *if* $\exists$ *a. h a* = *None then Some*(*LEAST a. h a* = *None*) *else None*

definition *cast-ok* :: *'m prog* $\Rightarrow$ *cname* $\Rightarrow$ *heap* $\Rightarrow$ *val* $\Rightarrow$ *bool*
where
cast-ok P C h v $\equiv$ *v* = *Null* $\vee$ *P* $\vdash$ *cname-of h* (*the-Addr v*) $\preceq^*$ *C*

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

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

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

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

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

lemma [*simp*]: (*typeof*$_h$ *v* = *Some*(*Class C*)) = $(\exists a\ fs.\ v$ = *Addr a* $\wedge$ *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 *Map.empty* because they do not contain addresses:

abbreviation
typeof :: *val* $\Rightarrow$ *ty option* **where**

typeof v == typeof-h Map.empty v

lemma *typeof-lit-typeof*:
typeof v = Some T ⟹ typeof$_h$ *v = Some T*

lemma *typeof-lit-is-type*:
typeof v = Some T ⟹ 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*: $[\![\ h \trianglelefteq h';\ typeof_h\ v = Some\ T\ ]\!] \Longrightarrow typeof_{h'}\ v = Some\ T$

Code generator setup for *new-Addr*

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

lemma *new-Addr-code-code* [*code*]:
new-Addr h = gen-new-Addr h 0
by(*simp add*: *new-Addr-def gen-new-Addr-def split del*: *if-split cong*: *if-cong*)

lemma *gen-new-Addr-code* [*code*]:
gen-new-Addr h n = (*if h n = None then Some n else gen-new-Addr h* (*Suc n*))
apply(*simp add*: *gen-new-Addr-def*)
apply(*rule impI*)
apply(*rule conjI*)
apply *safe*[*1*]
apply(*fastforce intro*: *Least-equality*)
apply(*rule arg-cong*[**where** *f*=*Least*])
apply(*rule ext*)
apply(*case-tac n = ac*)
apply *simp*
apply(*auto*)[*1*]
apply *clarify*
apply(*subgoal-tac a = n*)
apply *simp*
apply(*rule Least-equality*)
apply *auto*[*2*]
apply(*rule ccontr*)
apply(*erule-tac x*=*a* **in** *allE*)
apply *simp*
done

end

2.7 Exceptions

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

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

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

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

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

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

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

definition *preallocated* :: *heap* ⇒ *bool*
where
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 ⟹ *typeof*$_h$ (*Addr*(*addr-of-sys-xcpt OutOfMemory*)) = *Some*(*Class OutOfMemory*)

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

lemma *preallocated-hext*:
⟦ *preallocated h*; *h* ⊴ *h'* ⟧ ⟹ *preallocated h'*

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

end

2.8 Expressions

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

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

datatype *'a exp*
= *new cname* — class instance creation
| *Cast cname* (*'a exp*) — type cast
| *Val val* — value
| *BinOp* (*'a exp*) *bop* (*'a exp*) (- «-» - [*80*,*0*,*81*] *80*) — binary operation
| *Var 'a* — local variable (incl. parameter)
| *LAss 'a* (*'a exp*) (-:=- [*90*,*90*]*90*) — local assignment
| *FAcc* (*'a exp*) *vname cname* (-•-{-} [*10*,*90*,*99*]*90*) — field access
| *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*)

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

The semantics of binary operators:

fun *binop* :: *bop* × *val* × *val* ⇒ *val option* **where**

binop(*Eq*,v_1,v_2) = *Some*(*Bool* (v_1 = v_2))
| *binop*(*Add*,*Intg* i_1,*Intg* i_2) = *Some*(*Intg*(i_1+i_2))
| *binop*(*bop*,v_1,v_2) = *None*

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

2.8.1 Syntactic sugar

abbreviation (*input*)
InitBlock:: *'a* ⇒ *ty* ⇒ *'a exp* ⇒ *'a exp* ⇒ *'a exp* (((*1'*{-:- := -;/ -})) **where**
InitBlock V T e1 e2 == {*V*:*T*; *V* := *e1*;; *e2*}

abbreviation *unit* **where** *unit* == *Val Unit*
abbreviation *null* **where** *null* == *Val Null*
abbreviation *addr a* == *Val*(*Addr a*)
abbreviation *true* == *Val*(*Bool True*)
abbreviation *false* == *Val*(*Bool False*)

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

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

2.8.2 Free Variables

primrec *fv* :: *expr* ⇒ *vname set* **and** *fvs* :: *expr list* ⇒ *vname set* **where**
fv(*new C*) = {}
| *fv*(*Cast C e*) = *fv e*
| *fv*(*Val v*) = {}
| *fv*(e_1 «*bop*» e_2) = *fv* e_1 ∪ *fv* e_2
| *fv*(*Var V*) = {*V*}
| *fv*(*LAss V e*) = {*V*} ∪ *fv e*
| *fv*(*e*·*F*{*D*}) = *fv e*
| *fv*(e_1·*F*{*D*}:=e_2) = *fv* e_1 ∪ *fv* e_2
| *fv*(*e*·*M*(*es*)) = *fv e* ∪ *fvs es*
| *fv*({*V*:*T*; *e*}) = *fv e* − {*V*}
| *fv*(e_1;;e_2) = *fv* e_1 ∪ *fv* e_2
| *fv*(*if* (*b*) e_1 *else* e_2) = *fv b* ∪ *fv* e_1 ∪ *fv* e_2
| *fv*(*while* (*b*) *e*) = *fv b* ∪ *fv e*
| *fv*(*throw e*) = *fv e*
| *fv*(*try* e_1 *catch*(*C V*) e_2) = *fv* e_1 ∪ (*fv* e_2 − {*V*})
| *fvs*([]) = {}
| *fvs*(*e*#*es*) = *fv e* ∪ *fvs es*

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

2.9 Program State

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

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

definition *hp* :: *state* ⇒ *heap*
where
hp ≡ *fst*
definition *lcl* :: *state* ⇒ *locals*
where
lcl ≡ *snd*
end

2.10 Big Step Semantics

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

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

New:
⟦ *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*)⟩

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

| *WhileCondThrow*:
$P \vdash \langle e,s_0\rangle \Rightarrow \langle\ \mathit{throw}\ e',s_1\rangle \Longrightarrow$

$P \vdash \langle \mathit{while}\ (e)\ c,s_0\rangle \Rightarrow \langle \mathit{throw}\ e',s_1\rangle$

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

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

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

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

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

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

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

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

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

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

2.10.1 Final expressions

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

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

lemma [*simp*]: *final*(*Val v*)

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

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

lemma *eval-lcl-incr*: *P* ⊢ ⟨*e*,(*h*0,*l*0)⟩ ⇒ ⟨*e*′,(*h*1,*l*1)⟩ ⟹ *dom l*0 ⊆ *dom l*1
and *evals-lcl-incr*: *P* ⊢ ⟨*es*,(*h*0,*l*0)⟩ [⇒] ⟨*es*′,(*h*1,*l*1)⟩ ⟹ *dom l*0 ⊆ *dom l*1

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

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

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

theorem *eval-hext*: *P* ⊢ ⟨*e*,(*h*,*l*)⟩ ⇒ ⟨*e*′,(*h*′,*l*′)⟩ ⟹ *h* ⊴ *h*′
and *evals-hext*: *P* ⊢ ⟨*es*,(*h*,*l*)⟩ [⇒] ⟨*es*′,(*h*′,*l*′)⟩ ⟹ *h* ⊴ *h*′

end

2.11 Small Step Semantics

theory *SmallStep*
imports *Expr State*
begin

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

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

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

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

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

where

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

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

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

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

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

| *RedCast*:
$[\![\ hp\ s\ a = Some(D,fs);\ P \vdash D \preceq^*\ C\ ]\!]$
$\Longrightarrow P \vdash \langle Cast\ C\ (addr\ a),\ s\rangle \to \langle addr\ a,\ s\rangle$

| *RedCastFail*:
$[\![\ hp\ s\ a = Some(D,fs);\ \neg\ P \vdash D \preceq^*\ C\ ]\!]$
$\Longrightarrow P \vdash \langle Cast\ C\ (addr\ a),\ s\rangle \to \langle THROW\ ClassCast,\ s\rangle$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

— Exception propagation

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

2.11.1 The reflexive transitive closure

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

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

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

2.11.2 Some easy lemmas

lemma [*iff*]: ¬ $P \vdash \langle [],s\rangle\ [\to]\ \langle es',s'\rangle$
lemma [*iff*]: ¬ $P \vdash \langle Val\ v,s\rangle \to \langle e',s'\rangle$
lemma [*iff*]: ¬ $P \vdash \langle Throw\ a,s\rangle \to \langle e',s'\rangle$

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

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

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

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

end

2.12 System Classes

theory *SystemClasses*
imports *Decl Exceptions*
begin

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

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

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

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

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

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

end

2.13 Generic Well-formedness of programs

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

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

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

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

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

definition *wf-mdecl* :: *'m wf-mdecl-test* ⇒ *'m wf-mdecl-test*
where
wf-mdecl wf-md P C ≡ λ(*M*,*Ts*,*T*,*mb*).
(∀ *T*∈*set Ts*. *is-type P T*) ∧ *is-type P T* ∧ *wf-md P C* (*M*,*Ts*,*T*,*mb*)

definition *wf-cdecl* :: *'m wf-mdecl-test* ⇒ *'m prog* ⇒ *'m cdecl* ⇒ *bool*
where

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

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

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

2.13.1 Well-formedness lemmas

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

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

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

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

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

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

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

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

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

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

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

lemma *subcls-induct*:

$[\![$ *wf-prog wf-md P*; $\bigwedge C.\ \forall D.\ (C,D) \in (subcls1\ P)^+ \longrightarrow Q\ D \Longrightarrow Q\ C\ ]\!] \Longrightarrow Q\ C$

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

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

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

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

2.13.2 Well-formedness and method lookup

lemma *sees-wf-mdecl*:
assumes *wf*: *wf-prog wf-md P* **and** *sees*: $P \vdash C\ sees\ M{:}Ts\to T = m\ in\ D$
shows *wf-mdecl wf-md P D* (M,Ts,T,m)

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

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

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 $P \vdash C\ sees\ M{:}Ts\to T = m\ in\ C$

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

lemma *wf-prog-lift*:
assumes *wf*: *wf-prog* ($\lambda P\ C\ bd.\ A\ P\ C\ bd$) P
and *rule*:
$\bigwedge$*wf-md C M Ts C T m bd.*
wf-prog wf-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-prog* ($\lambda P\ C\ bd.\ B\ P\ C\ bd$) P

2.13.3 Well-formedness and field lookup

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

lemma *has-fields-types*:
⟦ $P \vdash C$ *has-fields FDTs*; (*FD*,*T*) $\in$ *set FDTs*; *wf-prog wf-md P* ⟧ $\Longrightarrow$ *is-type P T*

lemma *sees-field-is-type*:
⟦ $P \vdash C$ *sees F*:*T in D*; *wf-prog wf-md P* ⟧ $\Longrightarrow$ *is-type P T*
lemma *wf-syscls*:
set SystemClasses $\subseteq$ *set P* $\Longrightarrow$ *wf-syscls P*
end

2.14 Weak well-formedness of Jinja programs

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

definition *wwf-J-mdecl* :: *J-prog* $\Rightarrow$ *cname* $\Rightarrow$ *J-mb mdecl* $\Rightarrow$ *bool*
where
wwf-J-mdecl P C $\equiv$ λ(M,*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-J-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*)
abbreviation
wwf-J-prog :: *J-prog* $\Rightarrow$ *bool* **where**
wwf-J-prog == *wf-prog wwf-J-mdecl*

end

2.15 Equivalence of Big Step and Small Step Semantics

theory *Equivalence* **imports** *BigStep SmallStep WWellForm* **begin**

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

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*:
assumes e_1-*steps*: $P \vdash \langle e_1,s_0\rangle \to* \langle addr\ a,s_1\rangle$
and e_2-*steps*: $P \vdash \langle e_2,s_1\rangle \to* \langle Val\ v,(h_2,l_2)\rangle$
and *hC*: $Some(C,fs) = h_2\ a$
shows $P \vdash \langle e_1{\cdot}F\{D\}{:=}e_2,\ s_0\rangle \to* \langle unit,\ (h_2(a{\mapsto}(C,fs((F,D){\mapsto}v))),l_2)\rangle$
lemma *FAssRedsNull*:
assumes e_1-*steps*: $P \vdash \langle e_1,s_0\rangle \to* \langle null,s_1\rangle$
and e_2-*steps*: $P \vdash \langle e_2,s_1\rangle \to* \langle Val\ v,s_2\rangle$
shows $P \vdash \langle e_1{\cdot}F\{D\}{:=}e_2,\ s_0\rangle \to* \langle THROW\ NullPointer,\ s_2\rangle$
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*:
assumes e_1-*steps*: $P \vdash \langle e_1,s_0\rangle \to* \langle Val\ v,s_1\rangle$
and e_2-*steps*: $P \vdash \langle e_2,s_1\rangle \to* \langle throw\ e,s_2\rangle$
shows $P \vdash \langle e_1{\cdot}F\{D\}{:=}e_2,s_0\rangle \to* \langle throw\ e,s_2\rangle$

;;

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*:
assumes e_1-*steps*: $P \vdash \langle e_1,s_0\rangle \to* \langle Val\ v_1,s_1\rangle$
and e_2-*steps*: $P \vdash \langle e_2,s_1\rangle \to* \langle e_2{}',s_2\rangle$
shows $P \vdash \langle e_1;;e_2,\ s_0\rangle \to* \langle e_2{}',s_2\rangle$

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*:
assumes *e-steps*: $P \vdash \langle e,s_0\rangle \to* \langle true,s_1\rangle$
and e_1-*steps*: $P \vdash \langle e_1,\ s_1\rangle \to* \langle e',s_2\rangle$
shows $P \vdash \langle if\ (e)\ e_1\ else\ e_2,\ s_0\rangle \to* \langle e',s_2\rangle$
lemma *CondReds2F*:
assumes *e-steps*: $P \vdash \langle e,s_0\rangle \to* \langle false,s_1\rangle$
and e_2-*steps*: $P \vdash \langle e_2,\ s_1\rangle \to* \langle e',s_2\rangle$
shows $P \vdash \langle if\ (e)\ e_1\ else\ e_2,\ s_0\rangle \to* \langle e',s_2\rangle$

While

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

and *while-steps*: $P \vdash \langle \mathit{while}\ (b)\ c,s_2\rangle \to* \langle e,s_3\rangle$
shows $P \vdash \langle \mathit{while}\ (b)\ c,s_0\rangle \to* \langle e,s_3\rangle$
lemma *WhileTRedsThrow*:
assumes *b-steps*: $P \vdash \langle b,s_0\rangle \to* \langle \mathit{true},s_1\rangle$
and *c-steps*: $P \vdash \langle c,s_1\rangle \to* \langle \mathit{throw}\ e,s_2\rangle$
shows $P \vdash \langle \mathit{while}\ (b)\ c,s_0\rangle \to* \langle \mathit{throw}\ e,s_2\rangle$

Throw

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

InitBlock

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

Block

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

try-catch

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

List

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

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

Call

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

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

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

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

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

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

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

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

An now the actual method call reduction lemmas.

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

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

lemma *CallRedsFinal*:
assumes *wwf*: *wwf-J-prog P*
and $P \vdash \langle e,s_0\rangle \rightarrow* \langle addr\ a,s_1\rangle$
$P \vdash \langle es,s_1\rangle\ [\rightarrow]*\ \langle map\ Val\ vs,(h_2,l_2)\rangle$
$h_2\ a = Some(C,fs)\ P \vdash C\ sees\ M{:}Ts\rightarrow T = (pns,body)\ in\ D$
$size\ vs = size\ pns$
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*:
assumes *e-steps*: $P \vdash \langle e,s_0\rangle \to* \langle Val\ v,s_1\rangle$
and *es-steps*: $P \vdash \langle es,s_1\rangle\ [\to]* \langle map\ Val\ vs_1\ @\ throw\ a\ \#\ es_2,s_2\rangle$
shows $P \vdash \langle e{\cdot}M(es),s_0\rangle \to* \langle throw\ a,s_2\rangle$

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*:
assumes *e-steps*: $P \vdash \langle e,s_0\rangle \to* \langle null,s_1\rangle$
and *es-steps*: $P \vdash \langle es,s_1\rangle\ [\to]* \langle map\ Val\ vs,s_2\rangle$
shows $P \vdash \langle e{\cdot}M(es),s_0\rangle \to* \langle THROW\ NullPointer,s_2\rangle$

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*:
$P \vdash \langle e,(h,l)\rangle \Rightarrow \langle e',(h',l')\rangle \Longrightarrow (\bigwedge W.\ fv\ e \subseteq W \Longrightarrow P \vdash \langle e,(h,l|`W)\rangle \Rightarrow \langle e',(h',l'|`W)\rangle)$
and $P \vdash \langle es,(h,l)\rangle\ [\Rightarrow]\ \langle es',(h',l')\rangle \Longrightarrow (\bigwedge W.\ fvs\ es \subseteq W \Longrightarrow P \vdash \langle es,(h,l|`W)\rangle\ [\Rightarrow]\ \langle es',(h',l'|`W)\rangle)$

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*: $P \vdash \langle es,s\rangle\ [\to]* \langle es'',s''\rangle$ **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*: $[\![P \vdash \langle e,s\rangle \to* \langle e',s'\rangle; \mathit{final}\ e']\!] \Longrightarrow P \vdash \langle e,s\rangle \Rightarrow \langle e',s'\rangle$
and $[\![P \vdash \langle es,s\rangle\ [\to]* \langle es',s'\rangle; \mathit{finals}\ es']\!] \Longrightarrow P \vdash \langle es,s\rangle\ [\Rightarrow]\ \langle es',s'\rangle$

2.15.3 Equivalence

And now, the crowning achievement:

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

end

2.16 Well-typedness of Jinja expressions

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

type-synonym
env = *vname* $\rightharpoonup$ *ty*

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

WTNew:
is-class P C $\Longrightarrow$
P,*E* $\vdash$ *new C* :: *Class C*

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

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

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

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

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

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

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

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

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

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

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

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

| *WTThrow*:

$P,E \vdash e :: Class\ C \implies$
$P,E \vdash throw\ e :: Void$

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

— well-typed expression lists

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

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

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

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

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

2.17 Runtime Well-typedness

theory *WellTypeRT*
imports *WellType*
begin

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

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

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

| *WTrtCast*:
$[\![$ $P,E,h \vdash e$: *T*; *is-refT T*; *is-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*

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

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

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

| *WTrtCall*:
$[\![$ $P,E,h \vdash e$: *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*

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

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

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

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

— 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*]: $P,E,h \vdash Val\ v : T = (typeof_h\ v = Some\ T)$
lemma [*iff*]: $P,E,h \vdash Var\ v : T = (E\ v = Some\ T)$
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*]:
$\bigwedge Ts.\ (P,E,h \vdash map\ Val\ vs\ [:]\ Ts) = (map\ (typeof_h)\ vs = map\ Some\ Ts)$

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

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*: $P,E,h \vdash e : T \Longrightarrow h \trianglelefteq h' \Longrightarrow P,E,h' \vdash e : T$
and *WTrts-hext-mono*: $P,E,h \vdash es\ [:]\ Ts \Longrightarrow h \trianglelefteq h' \Longrightarrow P,E,h' \vdash es\ [:]\ Ts$

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* **imports** *BigStep* **begin**

2.18.1 Hypersets

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

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

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

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

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

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

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

lemma [*simp*]: $\lfloor \{\} \rfloor \sqcup A = A \ \wedge\ A \sqcup \lfloor \{\} \rfloor = A$
lemma [*simp*]: $\lfloor A \rfloor \sqcup \lfloor B \rfloor = \lfloor A \cup B \rfloor \wedge \lfloor A \rfloor \ominus a = \lfloor A - \{a\} \rfloor$
lemma [*simp*]: *None* $\sqcup A =$ *None* $\wedge A \sqcup$ *None* $=$ *None*
lemma [*simp*]: $a \in\in$ *None* $\wedge$ *None* $\ominus a =$ *None*
lemma *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

primrec
  $\mathcal{A}$ :: *'a exp* ⇒ *'a hyperset*
  and $\mathcal{A}s$ :: *'a exp list* ⇒ *'a hyperset*
where
  $\mathcal{A}$ (*new C*) = ⌊{}⌋
| $\mathcal{A}$ (*Cast C e*) = $\mathcal{A}$ *e*
| $\mathcal{A}$ (*Val v*) = ⌊{}⌋
| $\mathcal{A}$ (e_1 «*bop*» e_2) = $\mathcal{A}$ e_1 ⊔ $\mathcal{A}$ e_2
| $\mathcal{A}$ (*Var V*) = ⌊{}⌋
| $\mathcal{A}$ (*LAss V e*) = ⌊{*V*}⌋ ⊔ $\mathcal{A}$ *e*
| $\mathcal{A}$ (*e*·*F*{*D*}) = $\mathcal{A}$ *e*
| $\mathcal{A}$ (e_1·*F*{*D*}:=e_2) = $\mathcal{A}$ e_1 ⊔ $\mathcal{A}$ e_2
| $\mathcal{A}$ (*e*·*M*(*es*)) = $\mathcal{A}$ *e* ⊔ $\mathcal{A}s$ *es*
| $\mathcal{A}$ ({*V*:*T*; *e*}) = $\mathcal{A}$ *e* ⊖ *V*
| $\mathcal{A}$ (e_1;;e_2) = $\mathcal{A}$ e_1 ⊔ $\mathcal{A}$ e_2
| $\mathcal{A}$ (*if* (*e*) e_1 *else* e_2) = $\mathcal{A}$ *e* ⊔ ($\mathcal{A}$ e_1 ⊓ $\mathcal{A}$ 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 ⊓ ($\mathcal{A}$ e_2 ⊖ *V*)

| $\mathcal{A}s$ ([]) = ⌊{}⌋
| $\mathcal{A}s$ (*e*#*es*) = $\mathcal{A}$ *e* ⊔ $\mathcal{A}s$ *es*

primrec
  $\mathcal{D}$ :: *'a exp* ⇒ *'a hyperset* ⇒ *bool*
  and $\mathcal{D}s$ :: *'a exp list* ⇒ *'a hyperset* ⇒ *bool*
where
  $\mathcal{D}$ (*new C*) *A* = *True*
| $\mathcal{D}$ (*Cast C e*) *A* = $\mathcal{D}$ *e A*
| $\mathcal{D}$ (*Val v*) *A* = *True*
| $\mathcal{D}$ (e_1 «*bop*» e_2) *A* = ($\mathcal{D}$ e_1 *A* ∧ $\mathcal{D}$ e_2 (*A* ⊔ $\mathcal{A}$ e_1))
| $\mathcal{D}$ (*Var V*) *A* = (*V* ∈∈ *A*)
| $\mathcal{D}$ (*LAss V e*) *A* = $\mathcal{D}$ *e A*
| $\mathcal{D}$ (*e*·*F*{*D*}) *A* = $\mathcal{D}$ *e A*
| $\mathcal{D}$ (e_1·*F*{*D*}:=e_2) *A* = ($\mathcal{D}$ e_1 *A* ∧ $\mathcal{D}$ e_2 (*A* ⊔ $\mathcal{A}$ e_1))
| $\mathcal{D}$ (*e*·*M*(*es*)) *A* = ($\mathcal{D}$ *e A* ∧ $\mathcal{D}s$ *es* (*A* ⊔ $\mathcal{A}$ *e*))
| $\mathcal{D}$ ({*V*:*T*; *e*}) *A* = $\mathcal{D}$ *e* (*A* ⊖ *V*)
| $\mathcal{D}$ (e_1;;e_2) *A* = ($\mathcal{D}$ e_1 *A* ∧ $\mathcal{D}$ e_2 (*A* ⊔ $\mathcal{A}$ e_1))
| $\mathcal{D}$ (*if* (*e*) e_1 *else* e_2) *A* =
  ($\mathcal{D}$ *e A* ∧ $\mathcal{D}$ e_1 (*A* ⊔ $\mathcal{A}$ *e*) ∧ $\mathcal{D}$ e_2 (*A* ⊔ $\mathcal{A}$ *e*))
| $\mathcal{D}$ (*while* (*e*) *c*) *A* = ($\mathcal{D}$ *e A* ∧ $\mathcal{D}$ *c* (*A* ⊔ $\mathcal{A}$ *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* ∧ $\mathcal{D}$ e_2 (*A* ⊔ ⌊{*V*}⌋))

| $\mathcal{D}s$ ([]) *A* = *True*
| $\mathcal{D}s$ (*e*#*es*) *A* = ($\mathcal{D}$ *e A* ∧ $\mathcal{D}s$ *es* (*A* ⊔ $\mathcal{A}$ *e*))

lemma *As-map-Val*[*simp*]: $\mathcal{A}s$ (*map Val vs*) = ⌊{}⌋
lemma *D-append*[*iff*]: ⋀*A*. $\mathcal{D}s$ (*es* @ *es'*) *A* = ($\mathcal{D}s$ *es A* ∧ $\mathcal{D}s$ *es'* (*A* ⊔ $\mathcal{A}s$ *es*))

lemma *A-fv*: ⋀*A*. $\mathcal{A}$ *e* = ⌊*A*⌋ ⟹ *A* ⊆ *fv e*
and ⋀*A*. $\mathcal{A}s$ *es* = ⌊*A*⌋ ⟹ *A* ⊆ *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*
imports *Exceptions*
begin

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

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

definition *hconf* :: *'m prog* ⇒ *heap* ⇒ *bool* (- ⊢ - √ [51,51] 50)
where
$P \vdash h\ \surd \equiv$
$(\forall a\ obj.\ h\ a = Some\ obj \longrightarrow P,h \vdash obj\ \surd) \wedge preallocated\ h$

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

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

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*]: $P,h \vdash default\text{-}val\ T :\leq T$
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 Object conformance

lemma *oconf-hext*: $P,h \vdash obj\ \surd \Longrightarrow h \trianglelefteq h' \Longrightarrow P,h' \vdash obj\ \surd$
lemma *oconf-init-fields*:
$P \vdash C\ has\text{-}fields\ FDTs \Longrightarrow P,h \vdash (C,\ init\text{-}fields\ FDTs)\ \surd$
by(*fastforce simp add*: *has-field-def oconf-def init-fields-def map-of-map*
dest: *has-fields-fun*)

lemma *oconf-fupd* [*intro?*]:
$[\![\ P \vdash C\ has\ F{:}T\ in\ D;\ P,h \vdash v :\leq T;\ P,h \vdash (C,fs)\ \surd\ ]\!]$
$\Longrightarrow P,h \vdash (C,\ fs((F,D) \mapsto v))\ \surd$

2.19.4 Heap conformance

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

2.19.5 Local variable conformance

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

end

2.20 Progress of Small Step Semantics

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

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

Derivation of new induction scheme for well typing:

inductive
$\quad \textit{WTrt}' :: [\textit{J-prog},\textit{heap},\textit{env},\textit{expr},\textit{ty}] \Rightarrow \textit{bool}$
$\quad$ **and** $\textit{WTrts}' :: [\textit{J-prog},\textit{heap},\textit{env},\textit{expr list},\ \textit{ty list}] \Rightarrow \textit{bool}$
$\quad$ **and** $\textit{WTrt2}' :: [\textit{J-prog},\textit{env},\textit{heap},\textit{expr},\textit{ty}] \Rightarrow \textit{bool}$
$\qquad (\text{-},\text{-},\text{-} \vdash \text{-} :'' \text{-}\quad [51,51,51]50)$
$\quad$ **and** $\textit{WTrts2}' :: [\textit{J-prog},\textit{env},\textit{heap},\textit{expr list},\ \textit{ty list}] \Rightarrow \textit{bool}$
$\qquad (\text{-},\text{-},\text{-} \vdash \text{-}\ [:'']\ \text{-}\ [51,51,51]50)$
$\quad$ **for** $P :: \textit{J-prog}$ **and** $h :: \textit{heap}$
where
$\quad P,E,h \vdash e :' T \equiv \textit{WTrt}'\ P\ h\ E\ e\ T$
$|\ P,E,h \vdash es\ [:']\ Ts \equiv \textit{WTrts}'\ P\ h\ E\ es\ Ts$

$|\ \textit{is-class}\ P\ C \ \Longrightarrow\ P,E,h \vdash \textit{new}\ C :' \textit{Class}\ C$
$|\ [\![\ P,E,h \vdash e :' T;\ \textit{is-refT}\ T;\ \textit{is-class}\ P\ C\ ]\!]$
$\quad \Longrightarrow P,E,h \vdash \textit{Cast}\ C\ e :' \textit{Class}\ C$
$|\ \textit{typeof}_h\ v = \textit{Some}\ T \Longrightarrow P,E,h \vdash \textit{Val}\ v :' T$
$|\ E\ v = \textit{Some}\ T\ \Longrightarrow\ P,E,h \vdash \textit{Var}\ v :' T$
$|\ [\![\ P,E,h \vdash e_1 :' T_1;\ \ P,E,h \vdash e_2 :' T_2\ ]\!]$
$\quad \Longrightarrow P,E,h \vdash e_1\ «\textit{Eq}»\ e_2 :' \textit{Boolean}$
$|\ [\![\ P,E,h \vdash e_1 :' \textit{Integer};\ \ P,E,h \vdash e_2 :' \textit{Integer}\ ]\!]$
$\quad \Longrightarrow P,E,h \vdash e_1\ «\textit{Add}»\ e_2 :' \textit{Integer}$
$|\ [\![\ P,E,h \vdash \textit{Var}\ V :' T;\ \ P,E,h \vdash e :' T';\ \ P \vdash T' \leq T$ ~~$V \neq \textit{This}$~~ $]\!]$
$\quad \Longrightarrow P,E,h \vdash V{:=}e :' \textit{Void}$
$|\ [\![\ P,E,h \vdash e :' \textit{Class}\ C;\ P \vdash C\ \textit{has}\ F{:}T\ \textit{in}\ D\ ]\!] \Longrightarrow P,E,h \vdash e{\cdot}F\{D\} :' T$
$|\ P,E,h \vdash e :' \textit{NT} \Longrightarrow P,E,h \vdash e{\cdot}F\{D\} :' T$
$|\ [\![\ P,E,h \vdash e_1 :' \textit{Class}\ C;\ \ P \vdash C\ \textit{has}\ F{:}T\ \textit{in}\ D;$
$\qquad P,E,h \vdash e_2 :' T_2;\ \ P \vdash T_2 \leq T\ ]\!]$
$\quad \Longrightarrow P,E,h \vdash e_1{\cdot}F\{D\}{:=}e_2 :' \textit{Void}$
$|\ [\![\ P,E,h \vdash e_1{:}'\textit{NT};\ P,E,h \vdash e_2 :' T_2\ ]\!] \Longrightarrow P,E,h \vdash e_1{\cdot}F\{D\}{:=}e_2 :' \textit{Void}$
$|\ [\![\ P,E,h \vdash e :' \textit{Class}\ C;\ P \vdash C\ \textit{sees}\ M{:}Ts \rightarrow T = (\textit{pns},\textit{body})\ \textit{in}\ D;$
$\qquad P,E,h \vdash es\ [:']\ Ts';\ P \vdash Ts'\ [\leq]\ Ts\ ]\!]$
$\quad \Longrightarrow P,E,h \vdash e{\cdot}M(es) :' T$
$|\ [\![\ P,E,h \vdash e :' \textit{NT};\ P,E,h \vdash es\ [:']\ Ts\ ]\!] \Longrightarrow P,E,h \vdash e{\cdot}M(es) :' T$
$|\ P,E,h \vdash []\ [:']\ []$
$|\ [\![\ P,E,h \vdash e :' T;\ \ P,E,h \vdash es\ [:']\ Ts\ ]\!] \Longrightarrow\ \ P,E,h \vdash e\#es\ [:']\ T\#Ts$
$|\ [\![\ \textit{typeof}_h\ v = \textit{Some}\ T_1;\ P \vdash T_1 \leq T;\ P,E(V \mapsto T),h \vdash e_2 :' T_2\ ]\!]$
$\quad \Longrightarrow\ \ P,E,h \vdash \{V{:}T := \textit{Val}\ v;\ e_2\} :' T_2$
$|\ [\![\ P,E(V \mapsto T),h \vdash e :' T';\ \neg\ \textit{assigned}\ V\ e\ ]\!] \Longrightarrow\ \ P,E,h \vdash \{V{:}T;\ e\} :' T'$
$|\ [\![\ P,E,h \vdash e_1{:}'\ T_1;\ \ P,E,h \vdash e_2{:}'T_2\ ]\!]\ \ \Longrightarrow\ \ P,E,h \vdash e_1;;e_2 :' T_2$
$|\ [\![\ P,E,h \vdash e :' \textit{Boolean};\ \ P,E,h \vdash e_1{:}'\ T_1;\ \ P,E,h \vdash e_2{:}'\ T_2;$
$\qquad P \vdash T_1 \leq T_2 \vee P \vdash T_2 \leq T_1;$
$\qquad P \vdash T_1 \leq T_2 \longrightarrow T = T_2;\ P \vdash T_2 \leq T_1 \longrightarrow T = T_1\ ]\!]$
$\quad \Longrightarrow P,E,h \vdash \textit{if}\ (e)\ e_1\ \textit{else}\ e_2 :' T$

$|\ [\![\ P,E,h \vdash e :' \textit{Boolean};\ \ P,E,h \vdash c{:}'\ T\ ]\!]$
$\quad \Longrightarrow\ \ P,E,h \vdash \textit{while}(e)\ c :' \textit{Void}$
$|\ [\![\ P,E,h \vdash e :' T_r;\ \textit{is-refT}\ T_r\ ]\!]\ \ \Longrightarrow\ \ P,E,h \vdash \textit{throw}\ e :' T$

| ⟦ $P,E,h \vdash e_1 :' T_1$; $P,E(V \mapsto Class\ C),h \vdash e_2 :' T_2$; $P \vdash T_1 \leq T_2$ ⟧
$\Longrightarrow P,E,h \vdash try\ e_1\ catch(C\ V)\ e_2 :' T_2$

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

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* **and** *hconf*: $P \vdash h\ \surd$
shows *progress*: $P,E,h \vdash e : T \Longrightarrow$
$(\bigwedge l.\ ⟦\ \mathcal{D}\ e\ \lfloor dom\ l \rfloor;\ \neg\ final\ e\ ⟧ \Longrightarrow \exists e'\ s'.\ P \vdash \langle e,(h,l)\rangle \rightarrow \langle e',s'\rangle)$
and $P,E,h \vdash es\ [:]\ Ts \Longrightarrow$
$(\bigwedge l.\ ⟦\ \mathcal{D}s\ es\ \lfloor dom\ l \rfloor;\ \neg\ finals\ es\ ⟧ \Longrightarrow \exists es'\ s'.\ P \vdash \langle es,(h,l)\rangle\ [\rightarrow]\ \langle es',s'\rangle)$

end

2.21 Well-formedness Constraints

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

definition *wf-J-mdecl* :: *J-prog* ⇒ *cname* ⇒ *J-mb mdecl* ⇒ *bool*
where
wf-J-mdecl P C ≡ λ(*M*,*Ts*,*T*,(*pns*,*body*)).
length Ts = *length pns* ∧
distinct pns ∧
this ∉ *set pns* ∧
(∃ *T'*. *P*,[*this*↦*Class C*,*pns*[↦]*Ts*] ⊢ *body* :: *T'* ∧ *P* ⊢ *T'* ≤ *T*) ∧
$\mathcal{D}$ *body* ⌊{*this*} ∪ *set pns*⌋

lemma *wf-J-mdecl*[*simp*]:
wf-J-mdecl P C (*M*,*Ts*,*T*,*pns*,*body*) ≡
(*length Ts* = *length pns* ∧
distinct pns ∧
this ∉ *set pns* ∧
(∃ *T'*. *P*,[*this*↦*Class C*,*pns*[↦]*Ts*] ⊢ *body* :: *T'* ∧ *P* ⊢ *T'* ≤ *T*) ∧
$\mathcal{D}$ *body* ⌊{*this*} ∪ *set pns*⌋)

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

lemma *wf-J-prog-wf-J-mdecl*:
⟦ *wf-J-prog P*; (*C*, *D*, *fds*, *mths*) ∈ *set P*; *jmdcl* ∈ *set mths* ⟧

$\Longrightarrow$ *wf-J-mdecl P C jmdcl*

lemma *wf-mdecl-wwf-mdecl*: *wf-J-mdecl P C Md* $\Longrightarrow$ *wwf-J-mdecl P C Md*

lemma *wf-prog-wwf-prog*: *wf-J-prog P* $\Longrightarrow$ *wwf-J-prog P*

end

2.22 Type Safety Proof

theory *TypeSafe*
imports *Progress JWellForm*
begin

2.22.1 Basic preservation lemmas

First two easy preservation lemmas.

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

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

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

Now preservation of definite assignment.

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

Combining conformance of heap and local variables:

definition *sconf* :: *J-prog* $\Rightarrow$ *env* $\Rightarrow$ *state* $\Rightarrow$ *bool* (-,- $\vdash$ - $\surd$ [51,51,51]50)
where
$P,E \vdash s\ \surd \equiv$ *let* $(h,l) = s$ *in* $P \vdash h\ \surd \wedge P,h \vdash l\ (:\le)\ E$

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

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*: $P \vdash \langle e,(h,l)\rangle \to \langle e',(h',l')\rangle \Longrightarrow$
$(\bigwedge E\ T.\ [\![\ P,E \vdash (h,l)\ \surd;\ P,E,h \vdash e{:}T\ ]\!]$
$\Longrightarrow \exists\, T'.\ P,E,h' \vdash e'{:}T' \wedge P \vdash T' \leq T)$
and *subjects-reduction2*: $P \vdash \langle es,(h,l)\rangle\ [\to]\ \langle es',(h',l')\rangle \Longrightarrow$
$(\bigwedge E\ Ts.\ [\![\ P,E \vdash (h,l)\ \surd;\ P,E,h \vdash es\ [:]\ Ts\ ]\!]$
$\Longrightarrow \exists\, Ts'.\ P,E,h' \vdash es'\ [:]\ Ts' \wedge P \vdash Ts' [\leq] Ts)$

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

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 $!!T.\ [\![\ P,E \vdash s\surd;\ P,E,hp\ s \vdash e{:}T\ ]\!]$
$\Longrightarrow \exists\, T'.\ P \vdash T' \leq T \wedge P,E,hp\ s' \vdash e'{:}T'$

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 $[\![\ P \vdash \langle e,s\rangle \Rightarrow \langle e',s'\rangle;\ P,E \vdash s\surd;\ P,E \vdash e::T\ ]\!]$

$\Longrightarrow \exists\, T'.\ P \vdash T' \leq T \wedge P,E,hp\ s' \vdash e' : T'$

2.22.5 The final polish

The above preservation lemmas are now combined and packed nicely.

definition *wf-config* :: *J-prog* ⇒ *env* ⇒ *state* ⇒ *expr* ⇒ *ty* ⇒ *bool* (-,-,- ⊢ - : - √ [51,0,0,0,0]50)
where
$P,E,s \vdash e : T\ \surd \ \equiv\ P,E \vdash s\ \surd \wedge P,E,hp\ s \vdash e : T$

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 $[\![\ P,E,s \vdash e : T\ \surd; \mathcal{D}\ e\ \lfloor dom(lcl\ s)\rfloor; \neg\ final\ e\ ]\!] \Longrightarrow \exists\, e'\ s'.\ P \vdash \langle e,s\rangle \to \langle e',s'\rangle$

corollary *TypeSafety*:
fixes *s*::*state*
assumes *wf*: *wf-J-prog P* **and** *sconf*: $P,E \vdash s\ \surd$ **and** *wt*: $P,E \vdash e :: T$
and $\mathcal{D}$: $\mathcal{D}\ e\ \lfloor dom(lcl\ s)\rfloor$
and *steps*: $P \vdash \langle e,s\rangle \to* \langle e',s'\rangle$
and *nstep*: $\neg(\exists\, e''\ s''.\ P \vdash \langle e',s'\rangle \to \langle e'',s''\rangle)$
shows $(\exists\, v.\ e' = Val\ v \wedge P,hp\ s' \vdash v :\leq T) \vee$
$(\exists\, a.\ e' = Throw\ a \wedge a \in dom(hp\ s'))$

end

2.23 Program annotation

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

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

AnnoNew: $P,E \vdash new\ C \rightsquigarrow new\ C$
| *AnnoCast*: $P,E \vdash e \rightsquigarrow e' \Longrightarrow P,E \vdash Cast\ C\ e \rightsquigarrow Cast\ C\ e'$
| *AnnoVal*: $P,E \vdash Val\ v \rightsquigarrow Val\ v$
| *AnnoVarVar*: $E\ V = \lfloor T\rfloor \Longrightarrow P,E \vdash Var\ V \rightsquigarrow Var\ V$
| *AnnoVarField*: $[\![\ E\ V = None;\ E\ this = \lfloor Class\ C\rfloor;\ P \vdash C\ sees\ V:T\ in\ D\ ]\!]$
$\Longrightarrow P,E \vdash Var\ V \rightsquigarrow Var\ this{\cdot}V\{D\}$
| *AnnoBinOp*:
$[\![\ P,E \vdash e1 \rightsquigarrow e1';\ \ P,E \vdash e2 \rightsquigarrow e2'\ ]\!]$
$\Longrightarrow P,E \vdash e1\ «bop»\ e2 \rightsquigarrow e1'\ «bop»\ e2'$
| *AnnoLAssVar*:

$[\![E\ V = \lfloor T \rfloor;\ P,E \vdash e \rightsquigarrow e']\!] \Longrightarrow P,E \vdash V{:=}e \rightsquigarrow V{:=}e'$
| *AnnoLAssField*:
$[\![E\ V = None;\ E\ this = \lfloor Class\ C \rfloor;\ P \vdash C\ sees\ V{:}T\ in\ D;\ P,E \vdash e \rightsquigarrow e']\!]$
$\Longrightarrow P,E \vdash V{:=}e \rightsquigarrow Var\ this{\cdot}V\{D\} := e'$
| *AnnoFAcc*:
$[\![P,E \vdash e \rightsquigarrow e';\ P,E \vdash e' :: Class\ C;\ P \vdash C\ sees\ F{:}T\ in\ D]\!]$
$\Longrightarrow P,E \vdash e{\cdot}F\{[]\} \rightsquigarrow e'{\cdot}F\{D\}$
| *AnnoFAss*: $[\![P,E \vdash e1 \rightsquigarrow e1';\ P,E \vdash e2 \rightsquigarrow e2';$
$P,E \vdash e1' :: Class\ C;\ P \vdash C\ sees\ F{:}T\ in\ D]\!]$
$\Longrightarrow P,E \vdash e1{\cdot}F\{[]\} := e2 \rightsquigarrow e1'{\cdot}F\{D\} := e2'$
| *AnnoCall*:
$[\![P,E \vdash e \rightsquigarrow e';\ P,E \vdash es\ [\rightsquigarrow]\ es']\!]$
$\Longrightarrow P,E \vdash Call\ e\ M\ es \rightsquigarrow Call\ e'\ M\ es'$
| *AnnoBlock*:
$P,E(V \mapsto T) \vdash e \rightsquigarrow e' \Longrightarrow P,E \vdash \{V{:}T;\ e\} \rightsquigarrow \{V{:}T;\ e'\}$
| *AnnoComp*: $[\![P,E \vdash e1 \rightsquigarrow e1';\ P,E \vdash e2 \rightsquigarrow e2']\!]$
$\Longrightarrow P,E \vdash e1;;e2 \rightsquigarrow e1';;e2'$
| *AnnoCond*: $[\![P,E \vdash e \rightsquigarrow e';\ P,E \vdash e1 \rightsquigarrow e1';\ P,E \vdash e2 \rightsquigarrow e2']\!]$
$\Longrightarrow P,E \vdash if\ (e)\ e1\ else\ e2 \rightsquigarrow if\ (e')\ e1'\ else\ e2'$
| *AnnoLoop*: $[\![P,E \vdash e \rightsquigarrow e';\ P,E \vdash c \rightsquigarrow c']\!]$
$\Longrightarrow P,E \vdash while\ (e)\ c \rightsquigarrow while\ (e')\ c'$
| *AnnoThrow*: $P,E \vdash e \rightsquigarrow e' \Longrightarrow P,E \vdash throw\ e \rightsquigarrow throw\ e'$
| *AnnoTry*: $[\![P,E \vdash e1 \rightsquigarrow e1';\ P,E(V \mapsto Class\ C) \vdash e2 \rightsquigarrow e2']\!]$
$\Longrightarrow P,E \vdash try\ e1\ catch(C\ V)\ e2 \rightsquigarrow try\ e1'\ catch(C\ V)\ e2'$

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

end

2.24 Example Expressions

theory *Examples* **imports** *Expr* **begin**

definition *classObject*::*J-mb cdecl*
where
classObject == (''*Object*'','''',[],[])

definition *classI* :: *J-mb cdecl*
where
classI ==
(''*I*'', *Object*,
[],
[(''*mult*'',[*Integer*,*Integer*],*Integer*,[''*i*'',''*j*''],
if (*Var* ''*i*'' «*Eq*» *Val*(*Intg* 0)) (*Val*(*Intg* 0))
else Var ''*j*'' «*Add*»
Var this · ''*mult*''([*Var* ''*i*'' «*Add*» *Val*(*Intg* (− 1)),*Var* ''*j*''])
])

definition *classL* :: *J-mb cdecl*

where
classL ==
(''*L*'', *Object*,
[(''*F*'',*Integer*), (''*N*'',*Class* ''*L*'')],
[(''*app*'',[*Class* ''*L*''],*Void*,[''*l*'']],
if (*Var this* · ''*N*''{''*L*''} «*Eq*» *null*)
(*Var this* · ''*N*''{''*L*''} := *Var* ''*l*'')
else (*Var this* · ''*N*''{''*L*''}) · ''*app*''([*Var* ''*l*''])
])

definition *testExpr-BuildList* :: *expr*
where
testExpr-BuildList ==
{''*l1*'':*Class* ''*L*'' := *new* ''*L*'';
Var ''*l1*''·''*F*''{''*L*''} := *Val*(*Intg 1*);;
{''*l2*'':*Class* ''*L*'' := *new* ''*L*'';
Var ''*l2*''· ''*F*''{''*L*''} := *Val*(*Intg 2*);;
{''*l3*'':*Class* ''*L*'' := *new* ''*L*'';
Var ''*l3*''· ''*F*''{''*L*''} := *Val*(*Intg 3*);;
{''*l4*'':*Class* ''*L*'' := *new* ''*L*'';
Var ''*l4*''· ''*F*''{''*L*''} := *Val*(*Intg 4*);;
Var ''*l1*''·''*app*''([*Var* ''*l2*'']);;
Var ''*l1*''·''*app*''([*Var* ''*l3*'']);;
Var ''*l1*''·''*app*''([*Var* ''*l4*''])}}}}

definition *testExpr1* ::*expr*
where
testExpr1 == *Val*(*Intg 5*)
definition *testExpr2* ::*expr*
where
testExpr2 == *BinOp* (*Val*(*Intg 5*)) *Add* (*Val*(*Intg 6*))
definition *testExpr3* ::*expr*
where
testExpr3 == *BinOp* (*Var* ''*V*'') *Add* (*Val*(*Intg 6*))
definition *testExpr4* ::*expr*
where
testExpr4 == ''*V*'' := *Val*(*Intg 6*)
definition *testExpr5* ::*expr*
where
testExpr5 == *new* ''*Object*'';; {''*V*'':(*Class* ''*C*'') := *new* ''*C*''; *Var* ''*V*''·''*F*''{''*C*''} := *Val*(*Intg 42*)}
definition *testExpr6* ::*expr*
where
testExpr6 == {''*V*'':(*Class* ''*I*'') := *new* ''*I*''; *Var* ''*V*''·''*mult*''([*Val*(*Intg 40*),*Val*(*Intg 4*)])}

definition *mb-isNull*:: *expr*
where
mb-isNull == *Var this* · ''*test*''{''*A*''} «*Eq*» *null*

definition *mb-add*:: *expr*
where
mb-add == (*Var this* · ''*int*''{''*A*''} :=(*Var this* · ''*int*''{''*A*''} «*Add*» *Var* ''*i*''));; (*Var this* · ''*int*''{''*A*''})

definition *mb-mult-cond*:: *expr*
where
mb-mult-cond == (*Var* ''*j*'' «*Eq*» *Val* (*Intg 0*)) «*Eq*» *Val* (*Bool False*)

definition *mb-mult-block*:: *expr*
where
mb-mult-block == ''*temp*'':=(*Var* ''*temp*'' «*Add*» *Var* ''*i*'');;''*j*'':=(*Var* ''*j*'' «*Add*» *Val* (*Intg* (− *1*)))

definition *mb-mult*:: *expr*
where
mb-mult == {''*temp*'':*Integer*:=*Val* (*Intg 0*); *While* (*mb-mult-cond*) *mb-mult-block*;; (*Var this* · ''*int*''{''*A*''} := *Var* ''*temp*'';; *Var* ''*temp*'')}

definition *classA*:: *J-mb cdecl*
where
classA ==
(''*A*'', *Object*,
[(''*int*'',*Integer*),
(''*test*'',*Class* ''*A*'')],
[(''*isNull*'',[],*Boolean*,[], *mb-isNull*),
(''*add*'',[*Integer*],*Integer*,[''*i*''], *mb-add*),
(''*mult*'',[*Integer*,*Integer*],*Integer*,[''*i*'',''*j*''], *mb-mult*)])

definition *testExpr-ClassA*:: *expr*
where
testExpr-ClassA ==
{''*A1*'':*Class* ''*A*'':= *new* ''*A*'';
{''*A2*'':*Class* ''*A*'':= *new* ''*A*'';
{''*testint*'':*Integer*:= *Val* (*Intg 5*);
(*Var* ''*A2*''· ''*int*''{''*A*''} := (*Var* ''*A1*''· ''*add*''([*Var* ''*testint*'']));;
(*Var* ''*A2*''· ''*int*''{''*A*''} := (*Var* ''*A1*''· ''*add*''([*Var* ''*testint*'']));;
Var ''*A2*''· ''*mult*''([*Var* ''*A2*''· ''*int*''{''*A*''}, *Var* ''*testint*'']) }}}

end

2.25 Code Generation For BigStep

theory *execute-Bigstep*
imports
BigStep Examples
HOL−Library.Code-Target-Numeral
begin

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

inductive *map-val2* :: *expr list* ⇒ *val list* ⇒ *expr list* ⇒ *bool*
where
Nil: *map-val2* [] [] []

| *Cons*: *map-val2 xs ys zs* $\Longrightarrow$ *map-val2* (*Val y # xs*) (*y # ys*) *zs*
| *Throw*: *map-val2* (*throw e # xs*) [] (*throw e # xs*)

theorem *map-val-conv*: (*xs = map Val ys*) = *map-val xs ys*
theorem *map-val2-conv*:
(*xs = map Val ys @ throw e # zs*) = *map-val2 xs ys* (*throw e # zs*)
lemma *CallNull2*:
⟦ $P \vdash \langle e,s_0\rangle \Rightarrow \langle null,s_1\rangle$; $P \vdash \langle ps,s_1\rangle$ [⇒] $\langle evs,s_2\rangle$; *map-val evs vs* ⟧
$\Longrightarrow P \vdash \langle e{\cdot}M(ps),s_0\rangle \Rightarrow \langle THROW\ NullPointer,s_2\rangle$
apply(*rule CallNull*, *assumption*+)
apply(*simp add*: *map-val-conv*[*symmetric*])
done

lemma *CallParamsThrow2*:
⟦ $P \vdash \langle e,s_0\rangle \Rightarrow \langle Val\ v,s_1\rangle$; $P \vdash \langle es,s_1\rangle$ [⇒] $\langle evs,s_2\rangle$;
map-val2 evs vs (*throw ex # es′*) ⟧
$\Longrightarrow P \vdash \langle e{\cdot}M(es),s_0\rangle \Rightarrow \langle throw\ ex,s_2\rangle$
apply(*rule eval-evals.CallParamsThrow*, *assumption*+)
apply(*simp add*: *map-val2-conv*[*symmetric*])
done

lemma *Call2*:
⟦ $P \vdash \langle e,s_0\rangle \Rightarrow \langle addr\ a,s_1\rangle$; $P \vdash \langle ps,s_1\rangle$ [⇒] $\langle evs,(h_2,l_2)\rangle$;
map-val evs vs;
$h_2\ a = Some(C,fs)$; $P \vdash C\ sees\ M{:}Ts\rightarrow T = (pns,body)\ in\ D$;
length vs = length pns; $l_2{'} = [this\mapsto Addr\ a,\ pns[\mapsto]vs]$;
$P \vdash \langle body,(h_2,l_2{'})\rangle \Rightarrow \langle e',(h_3,l_3)\rangle$ ⟧
$\Longrightarrow P \vdash \langle e{\cdot}M(ps),s_0\rangle \Rightarrow \langle e',(h_3,l_2)\rangle$
apply(*rule Call*, *assumption*+)
apply(*simp add*: *map-val-conv*[*symmetric*])
apply *assumption*+
done

code-pred
(*modes*: *i* ⇒ *o* ⇒ *bool*)
map-val
.

code-pred
(*modes*: *i* ⇒ *o* ⇒ *o* ⇒ *bool*)
map-val2
.

lemmas [*code-pred-intro*] =
eval-evals.New eval-evals.NewFail
eval-evals.Cast eval-evals.CastNull eval-evals.CastFail eval-evals.CastThrow
eval-evals.Val eval-evals.Var
eval-evals.BinOp eval-evals.BinOpThrow1 eval-evals.BinOpThrow2
eval-evals.LAss eval-evals.LAssThrow
eval-evals.FAcc eval-evals.FAccNull eval-evals.FAccThrow
eval-evals.FAss eval-evals.FAssNull
eval-evals.FAssThrow1 eval-evals.FAssThrow2
eval-evals.CallObjThrow

declare *CallNull2* [*code-pred-intro CallNull2*]
declare *CallParamsThrow2* [*code-pred-intro CallParamsThrow2*]
declare *Call2* [*code-pred-intro Call2*]

lemmas [*code-pred-intro*] =
eval-evals.Block
eval-evals.Seq eval-evals.SeqThrow
eval-evals.CondT eval-evals.CondF eval-evals.CondThrow
eval-evals.WhileF eval-evals.WhileT
eval-evals.WhileCondThrow

declare *eval-evals.WhileBodyThrow* [*code-pred-intro WhileBodyThrow2*]

lemmas [*code-pred-intro*] =
eval-evals.Throw eval-evals.ThrowNull
eval-evals.ThrowThrow
eval-evals.Try eval-evals.TryCatch eval-evals.TryThrow
eval-evals.Nil eval-evals.Cons eval-evals.ConsThrow

code-pred
(*modes*: *i* ⇒ *i* ⇒ *i* ⇒ *o* ⇒ *o* ⇒ *bool as execute*)
eval
proof −
case *eval*
from *eval.prems* **show** *thesis*
proof(*cases* (*no-simp*))
case *CallNull* **thus** *?thesis*
by(*rule eval.CallNull2*[*OF refl*])(*simp add*: *map-val-conv*[*symmetric*])
next
case *CallParamsThrow* **thus** *?thesis*
by(*rule eval.CallParamsThrow2*[*OF refl*])(*simp add*: *map-val2-conv*[*symmetric*])
next
case *Call* **thus** *?thesis*
by −(*rule eval.Call2*[*OF refl*], *simp-all add*: *map-val-conv*[*symmetric*])
next
case *WhileBodyThrow* **thus** *?thesis* **by**(*rule eval.WhileBodyThrow2*[*OF refl*])
qed(*assumption*|*erule* (*4*) *eval.that*[*OF refl*]|*erule* (*3*) *eval.that*[*OF refl*])+
next
case *evals*
from *evals.prems* **show** *thesis*
by(*cases* (*no-simp*))(*assumption*|*erule* (*3*) *evals.that*[*OF refl*])+
qed

notation *execute* (- ⊢ ((*1*⟨-,/-⟩) ⇒/ ⟨*'*-, *'*-⟩) [*51*,*0*,*0*] *81*)

definition *test1* = [] ⊢ ⟨*testExpr1*,(*Map.empty*,*Map.empty*)⟩ ⇒ ⟨-,-⟩
definition *test2* = [] ⊢ ⟨*testExpr2*,(*Map.empty*,*Map.empty*)⟩ ⇒ ⟨-,-⟩
definition *test3* = [] ⊢ ⟨*testExpr3*,(*Map.empty*,*Map.empty*(*''V''*↦*Intg 77*))⟩ ⇒ ⟨-,-⟩
definition *test4* = [] ⊢ ⟨*testExpr4*,(*Map.empty*,*Map.empty*)⟩ ⇒ ⟨-,-⟩
definition *test5* = [(*''Object''*,(*''''*,[],[])),(*''C''*,(*''Object''*,[(*''F''*,*Integer*)],[]))] ⊢ ⟨*testExpr5*,(*Map.empty*,*Map.empty*
⇒ ⟨-,-⟩
definition *test6* = [(*''Object''*,(*''''*,[],[])), *classI*] ⊢ ⟨*testExpr6*,(*Map.empty*,*Map.empty*)⟩ ⇒ ⟨-,-⟩

definition $V = ''V''$
definition $C = ''C''$
definition $F = ''F''$

ML-val ‹
val SOME ((@{code Val} (@{code Intg} (@{code int-of-integer} 5)), -), -) = Predicate.yield @{code test1};
val SOME ((@{code Val} (@{code Intg} (@{code int-of-integer} 11)), -), -) = Predicate.yield @{code test2};
val SOME ((@{code Val} (@{code Intg} (@{code int-of-integer} 83)), -), -) = Predicate.yield @{code test3};

val SOME ((-, (-, l)), -) = Predicate.yield @{code test4};
val SOME (@{code Intg} (@{code int-of-integer} 6)) = l @{code V};

val SOME ((-, (h, -)), -) = Predicate.yield @{code test5};
val SOME (c, fs) = h (@{code nat-of-integer} 1);
val SOME (obj, -) = h (@{code nat-of-integer} 0);
val SOME (@{code Intg} i) = fs (@{code F}, @{code C});
@{assert} (c = @{code C} andalso obj = @{code Object} andalso i = @{code int-of-integer} 42);

val SOME ((@{code Val} (@{code Intg} (@{code int-of-integer} 160)), -), -) = Predicate.yield @{code test6};
›

definition *test7* = [*classObject*, *classL*] ⊢ ⟨*testExpr-BuildList*, (*Map.empty*,*Map.empty*)⟩ ⇒ ⟨-,-⟩

definition $L = ''L''$
definition $N = ''N''$

ML-val ‹
val SOME ((-, (h, -)), -) = Predicate.yield @{code test7};
val SOME (-, fs1) = h (@{code nat-of-integer} 0);
val SOME (-, fs2) = h (@{code nat-of-integer} 1);
val SOME (-, fs3) = h (@{code nat-of-integer} 2);
val SOME (-, fs4) = h (@{code nat-of-integer} 3);

val F = @{code F};
val L = @{code L};
val N = @{code N};

@{assert} (fs1 (F, L) = SOME (@{code Intg} (@{code int-of-integer} 1)) andalso
fs1 (N, L) = SOME (@{code Addr} (@{code nat-of-integer} 1)) andalso
fs2 (F, L) = SOME (@{code Intg} (@{code int-of-integer} 2)) andalso
fs2 (N, L) = SOME (@{code Addr} (@{code nat-of-integer} 2)) andalso
fs3 (F, L) = SOME (@{code Intg} (@{code int-of-integer} 3)) andalso
fs3 (N, L) = SOME (@{code Addr} (@{code nat-of-integer} 3)) andalso
fs4 (F, L) = SOME (@{code Intg} (@{code int-of-integer} 4)) andalso
fs4 (N, L) = SOME @{code Null});
›

definition *test8* = [*classObject*, *classA*] ⊢ ⟨*testExpr-ClassA*, (*Map.empty*,*Map.empty*)⟩ ⇒ ⟨-,-⟩
definition $i = ''int''$
definition $t = ''test''$

definition $A = ''A''$

ML-val ‹
val SOME ((-, (h, l)), -) = Predicate.yield @{code test8};
val SOME (-, fs1) = h (@{code nat-of-integer} 0);
val SOME (-, fs2) = h (@{code nat-of-integer} 1);

val i = @{code i};
val t = @{code t};
val A = @{code A};

@{assert} (fs1 (i, A) = SOME (@{code Intg} (@{code int-of-integer} 10)) andalso
fs1 (t, A) = SOME @{code Null} andalso
fs2 (i, A) = SOME (@{code Intg} (@{code int-of-integer} 50)) andalso
fs2 (t, A) = SOME @{code Null});
›

end

2.26 Code Generation For WellType

theory *execute-WellType*
imports
WellType Examples
begin

lemma *WTCond1*:
$[\![P,E \vdash e :: Boolean;\ P,E \vdash e_1::T_1;\ P,E \vdash e_2::T_2;\ P \vdash T_1 \leq T_2;$
$P \vdash T_2 \leq T_1 \longrightarrow T_2 = T_1]\!] \Longrightarrow P,E \vdash if\ (e)\ e_1\ else\ e_2 :: T_2$
by (*fastforce*)

lemma *WTCond2*:
$[\![P,E \vdash e :: Boolean;\ P,E \vdash e_1::T_1;\ P,E \vdash e_2::T_2;\ P \vdash T_2 \leq T_1;$
$P \vdash T_1 \leq T_2 \longrightarrow T_1 = T_2]\!] \Longrightarrow P,E \vdash if\ (e)\ e_1\ else\ e_2 :: T_1$
by (*fastforce*)

lemmas [*code-pred-intro*] =
WT-WTs.WTNew
WT-WTs.WTCast
WT-WTs.WTVal
WT-WTs.WTVar
WT-WTs.WTBinOpEq
WT-WTs.WTBinOpAdd
WT-WTs.WTLAss
WT-WTs.WTFAcc
WT-WTs.WTFAss
WT-WTs.WTCall
WT-WTs.WTBlock
WT-WTs.WTSeq

declare

```
  WTCond1 [code-pred-intro WTCond1]
  WTCond2 [code-pred-intro WTCond2]

lemmas [code-pred-intro] =
  WT-WTs.WTWhile
  WT-WTs.WTThrow
  WT-WTs.WTTry
  WT-WTs.WTNil
  WT-WTs.WTCons

code-pred
  (modes: i ⇒ i ⇒ i ⇒ i ⇒ bool as type-check, i ⇒ i ⇒ i ⇒ o ⇒ bool as infer-type)
  WT
proof −
  case WT
  from WT.prems show thesis
  proof(cases (no-simp))
    case (WTCond E e e1 T1 e2 T2 T)
    from ‹x ⊢ T1 ≤ T2 ∨ x ⊢ T2 ≤ T1› show thesis
    proof
      assume x ⊢ T1 ≤ T2
      with ‹x ⊢ T1 ≤ T2 ⟶ T = T2› have T = T2 ..
      from ‹xa = E› ‹xb = if (e) e1 else e2› ‹xc = T› ‹x,E ⊢ e :: Boolean›
        ‹x,E ⊢ e1 :: T1› ‹x,E ⊢ e2 :: T2› ‹x ⊢ T1 ≤ T2› ‹x ⊢ T2 ≤ T1 ⟶ T = T1›
      show ?thesis unfolding ‹T = T2› by(rule WT.WTCond1[OF refl])
    next
      assume x ⊢ T2 ≤ T1
      with ‹x ⊢ T2 ≤ T1 ⟶ T = T1› have T = T1 ..
      from ‹xa = E› ‹xb = if (e) e1 else e2› ‹xc = T› ‹x,E ⊢ e :: Boolean›
        ‹x,E ⊢ e1 :: T1› ‹x,E ⊢ e2 :: T2› ‹x ⊢ T2 ≤ T1› ‹x ⊢ T1 ≤ T2 ⟶ T = T2›
      show ?thesis unfolding ‹T = T1› by(rule WT.WTCond2[OF refl])
    qed
  qed(assumption|erule (2) WT.that[OF refl])+
next
  case WTs
  from WTs.prems show thesis
    by(cases (no-simp))(assumption|erule (2) WTs.that[OF refl])+
qed

notation infer-type (-,- ⊢ - :: '- [51,51,51]100)

definition test1 where test1 = [],Map.empty ⊢ testExpr1 :: -
definition test2 where test2 = [], Map.empty ⊢ testExpr2 :: -
definition test3 where test3 = [], Map.empty(''V'' ↦ Integer) ⊢ testExpr3 :: -
definition test4 where test4 = [], Map.empty(''V'' ↦ Integer) ⊢ testExpr4 :: -
definition test5 where test5 = [classObject, (''C'',(''Object'',[(''F'',Integer)],[]))], Map.empty ⊢ tes-
tExpr5 :: -
definition test6 where test6 = [classObject, classI], Map.empty ⊢ testExpr6 :: -

ML-val ‹
  val SOME(@{code Integer}, -) = Predicate.yield @{code test1};
  val SOME(@{code Integer}, -) = Predicate.yield @{code test2};
  val SOME(@{code Integer}, -) = Predicate.yield @{code test3};
  val SOME(@{code Void}, -) = Predicate.yield @{code test4};
```

val SOME(@{*code Void*}, -) = *Predicate.yield* @{*code test5*};
val SOME(@{*code Integer*}, -) = *Predicate.yield* @{*code test6*};
›

definition *testmb-isNull* **where** *testmb-isNull* = [*classObject*, *classA*], *Map.empty*([*this*] [↦] [*Class ''A''*]) ⊢ *mb-isNull* :: -
definition *testmb-add* **where** *testmb-add* = [*classObject*, *classA*], *Map.empty*([*this*,*''i''*] [↦] [*Class ''A''*,*Integer*]) ⊢ *mb-add* :: -
definition *testmb-mult-cond* **where** *testmb-mult-cond* = [*classObject*, *classA*], *Map.empty*([*this*,*''j''*] [↦] [*Class ''A''*,*Integer*]) ⊢ *mb-mult-cond* :: -
definition *testmb-mult-block* **where** *testmb-mult-block* = [*classObject*, *classA*], *Map.empty*([*this*,*''i''*,*''j''*,*''temp''*] [↦] [*Class ''A''*,*Integer*,*Integer*,*Integer*]) ⊢ *mb-mult-block* :: -
definition *testmb-mult* **where** *testmb-mult* = [*classObject*, *classA*], *Map.empty*([*this*,*''i''*,*''j''*] [↦] [*Class ''A''*,*Integer*,*Integer*]) ⊢ *mb-mult* :: -

ML-val ‹
val SOME(@{*code Boolean*}, -) = *Predicate.yield* @{*code testmb-isNull*};
val SOME(@{*code Integer*}, -) = *Predicate.yield* @{*code testmb-add*};
val SOME(@{*code Boolean*}, -) = *Predicate.yield* @{*code testmb-mult-cond*};
val SOME(@{*code Void*}, -) = *Predicate.yield* @{*code testmb-mult-block*};
val SOME(@{*code Integer*}, -) = *Predicate.yield* @{*code testmb-mult*};
›

definition *test* **where** *test* = [*classObject*, *classA*], *Map.empty* ⊢ *testExpr-ClassA* :: -

ML-val ‹
val SOME(@{*code Integer*}, -) = *Predicate.yield* @{*code test*};
›

end

Chapter 3

Jinja Virtual Machine

3.1 State of the JVM

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

3.1.1 Frame Stack

type-synonym
pc = *nat*

type-synonym
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

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

end

3.2 Instructions of the JVM

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

datatype
instr = *Load nat* — load from local variable
| *Store nat* — store into local variable
| *Push val* — push a value (constant)
| *New cname* — create object
| *Getfield vname cname* — Fetch field from object
| *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

type-synonym
bytecode = *instr list*

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

type-synonym
ex-table = *ex-entry list*

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

type-synonym
jvm-prog = *jvm-method prog*

end

3.3 JVM Instruction Semantics

theory *JVMExecInstr*
imports *JVMInstructions JVMState ../Common/Exceptions*
begin

primrec
exec-instr :: [*instr*, *jvm-prog*, *heap*, *val list*, *val list*,
cname, *mname*, *pc*, *frame list*] => *jvm-state*
where
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, C₀, M₀, pc)#frs)
  | Some a ⇒ (None, h(a↦blank P C), (Addr a#stk, loc, C₀, M₀, pc+1)#frs))

| exec-instr (Getfield F C) P h stk loc C₀ M₀ 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, C₀, M₀, pc+1)#frs))

| exec-instr (Putfield F C) P h stk loc C₀ M₀ 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, C₀, M₀, pc+1)#frs))

| exec-instr (Checkcast C) P h stk loc C₀ M₀ 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, C₀, M₀, pc+1)#frs))

| exec-instr-Invoke:
 exec-instr (Invoke M n) P h stk loc C₀ M₀ 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,mxl₀,ins,xt)= method P C M;
       f'  = ([],[r]@(rev ps)@(replicate mxl₀ undefined),D,M,0)
   in (xp', h, f'#(stk, loc, C₀, M₀, pc)#frs))

| exec-instr Return P h stk₀ loc₀ C₀ M₀ pc frs =
  (if frs=[] then (None, h, []) else
   let v = hd stk₀;
       (stk,loc,C,m,pc) = hd frs;
       n = length (fst (snd (method P C₀ M₀)))
   in (None, h, (v#(drop (n+1) stk),loc,C,m,pc+1)#tl frs))

| exec-instr Pop P h stk loc C₀ M₀ pc frs =
     (None, h, (tl stk, loc, C₀, M₀, pc+1)#frs)

| exec-instr IAdd P h stk loc C₀ M₀ pc frs =
  (let i₂ = the-Intg (hd stk);
       i₁ = the-Intg (hd (tl stk))
   in (None, h, (Intg (i₁+i₂)#(tl (tl stk)), loc, C₀, M₀, pc+1)#frs))

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

| exec-instr CmpEq P h stk loc C₀ M₀ 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* ⌊*addr-of-sys-xcpt NullPointer*⌋ *else* ⌊*the-Addr*(*hd stk*)⌋
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* ⌊*addr-of-sys-xcpt NullPointer*⌋ *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* ⌊*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'*, (*stk*, *loc*, C_0, M_0, *pc+1*)#*frs*))
by *simp*

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

lemma *exec-instr-Return*:
exec-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-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-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-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-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-instr Throw P h (v#stk) loc C_0 M_0 pc frs =
(let xp′ = if v = Null then ⌊addr-of-sys-xcpt NullPointer⌋ else ⌊the-Addr v⌋
in (xp′, h, (v#stk, loc, C_0, M_0, pc)#frs))
by *simp*

end

3.4 Exception handling in the JVM

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

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

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

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

primrec *find-handler* :: *jvm-prog ⇒ addr ⇒ heap ⇒ frame list ⇒ jvm-state*
where
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*
imports *JVMExecInstr JVMExceptions*
begin

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

fun *exec* :: *jvm-prog* × *jvm-state* => *jvm-state option* **where** — single step execution
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*

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

— reflexive transitive closure:
definition *exec-all* :: *jvm-prog* ⇒ *jvm-state* ⇒ *jvm-state* ⇒ *bool*
((- ⊢/ - −*jvm*→/ -) [*61*,*61*,*61*]*60*) **where**

exec-all-def1: *P* ⊢ *σ* −*jvm*→ *σ'* ⟷ (*σ*,*σ'*) ∈ (*exec-1 P*)*

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

lemma *exec-1-eq*:
exec-1 P = {(*σ*,*σ'*). *exec* (*P*,*σ*) = *Some σ'*}
lemma *exec-1-iff*:
P ⊢ *σ* −*jvm*→$_1$ *σ'* = (*exec* (*P*,*σ*) = *Some σ'*)
lemma *exec-all-def*:
P ⊢ *σ* −*jvm*→ *σ'* = ((*σ*,*σ'*) ∈ {(*σ*,*σ'*). *exec* (*P*,*σ*) = *Some σ'*}*)

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.

definition *start-state* :: *jvm-prog* ⇒ *cname* ⇒ *mname* ⇒ *jvm-state* **where**
start-state P C M =
(*let* (*D*,*Ts*,*T*,*mxs*,mxl_0,*b*) = *method P C M in*
(*None*, *start-heap P*, [([], *Null* # *replicate* mxl_0 *undefined*, *C*, *M*, *0*)]))

end

3.6 A Defensive JVM

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

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

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

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

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

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

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

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

```
    (n < length loc)

| check-instr-Store:
    check-instr (Store n) P h stk loc C₀ M₀ pc frs =
    (0 < length stk ∧ n < length loc)

| check-instr-Push:
    check-instr (Push v) P h stk loc C₀ M₀ pc frs =
    (¬is-Addr v)

| check-instr-New:
    check-instr (New C) P h stk loc C₀ M₀ pc frs =
    is-class P C

| check-instr-Getfield:
    check-instr (Getfield F C) P h stk loc C₀ M₀ pc frs =
    (0 < length stk ∧ (∃ C' T. P ⊢ C sees F:T in C') ∧
    (let (C', T) = field P C F; ref = hd stk in
      C' = C ∧ is-Ref ref ∧ (ref ≠ Null ⟶
        h (the-Addr ref) ≠ None ∧
        (let (D,vs) = the (h (the-Addr ref)) in
          P ⊢ D ≼* C ∧ vs (F,C) ≠ None ∧ P,h ⊢ the (vs (F,C)) :≤ T))))

| check-instr-Putfield:
    check-instr (Putfield F C) P h stk loc C₀ M₀ pc frs =
    (1 < length stk ∧ (∃ C' T. P ⊢ C sees F:T in C') ∧
    (let (C', T) = field P C F; v = hd stk; ref = hd (tl stk) in
      C' = C ∧ is-Ref ref ∧ (ref ≠ Null ⟶
        h (the-Addr ref) ≠ None ∧
        (let D = fst (the (h (the-Addr ref))) in
          P ⊢ D ≼* C ∧ P,h ⊢ v :≤ T))))

| check-instr-Checkcast:
    check-instr (Checkcast C) P h stk loc C₀ M₀ pc frs =
    (0 < length stk ∧ is-class P C ∧ is-Ref (hd stk))

| check-instr-Invoke:
    check-instr (Invoke M n) P h stk loc C₀ M₀ pc frs =
    (n < length stk ∧ is-Ref (stk!n) ∧
    (stk!n ≠ Null ⟶
      (let a = the-Addr (stk!n);
           C = cname-of h a;
           Ts = fst (snd (method P C M))
      in h a ≠ None ∧ P ⊢ C has M ∧
         P,h ⊢ rev (take n stk) [:≤] Ts)))

| check-instr-Return:
    check-instr Return P h stk loc C₀ M₀ pc frs =
    (0 < length stk ∧ ((0 < length frs) ⟶
      (P ⊢ C₀ has M₀) ∧
      (let v = hd stk;
           T = fst (snd (snd (method P C₀ M₀)))
       in P,h ⊢ v :≤ T)))
```

| *check-instr-Pop*:
  check-instr Pop P h stk loc C_0 M_0 *pc frs* =
  (*0* < *length stk*)

| *check-instr-IAdd*:
  check-instr IAdd P h stk loc C_0 M_0 *pc frs* =
  (*1* < *length stk* ∧ *is-Intg* (*hd stk*) ∧ *is-Intg* (*hd* (*tl stk*)))

| *check-instr-IfFalse*:
  check-instr (*IfFalse b*) *P h stk loc* C_0 M_0 *pc frs* =
  (*0* < *length stk* ∧ *is-Bool* (*hd stk*) ∧ *0* ≤ *int pc*+*b*)

| *check-instr-CmpEq*:
  check-instr CmpEq P h stk loc C_0 M_0 *pc frs* =
  (*1* < *length stk*)

| *check-instr-Goto*:
  check-instr (*Goto b*) *P h stk loc* C_0 M_0 *pc frs* =
  (*0* ≤ *int pc*+*b*)

| *check-instr-Throw*:
  check-instr Throw P h stk loc C_0 M_0 *pc frs* =
  (*0* < *length stk* ∧ *is-Ref* (*hd stk*))

definition *check* :: *jvm-prog* ⇒ *jvm-state* ⇒ *bool* **where**
  check P σ = (*let* (*xcpt*, *h*, *frs*) = σ *in*
  (*case frs of* [] ⇒ *True* | (*stk*,*loc*,*C*,*M*,*pc*)#*frs*′ ⇒
  P ⊢ *C has M* ∧
  (*let* (*C*′,*Ts*,*T*,*mxs*,*mxl*$_0$,*ins*,*xt*) = *method P C M*; *i* = *ins*!*pc in*
  pc < *size ins* ∧ *size stk* ≤ *mxs* ∧
  check-instr i P h stk loc C M pc frs′)))

definition *exec-d* :: *jvm-prog* ⇒ *jvm-state* ⇒ *jvm-state option type-error* **where**
  exec-d P σ = (*if check P* σ *then Normal* (*exec* (*P*, σ)) *else TypeError*)

inductive-set
  exec-1-d :: *jvm-prog* ⇒ (*jvm-state type-error* × *jvm-state type-error*) *set*
  and *exec-1-d*′ :: *jvm-prog* ⇒ *jvm-state type-error* ⇒ *jvm-state type-error* ⇒ *bool*
  (- ⊢ - −*jvmd*→$_1$ - [*61*,*61*,*61*]*60*)
  for *P* :: *jvm-prog*
where
  P ⊢ σ −*jvmd*→$_1$ σ′ ≡ (σ,σ′) ∈ *exec-1-d P*
| *exec-1-d-ErrorI*: *exec-d P* σ = *TypeError* ⟹ *P* ⊢ *Normal* σ −*jvmd*→$_1$ *TypeError*
| *exec-1-d-NormalI*: *exec-d P* σ = *Normal* (*Some* σ′) ⟹ *P* ⊢ *Normal* σ −*jvmd*→$_1$ *Normal* σ′

— reflexive transitive closure:
definition *exec-all-d* :: *jvm-prog* ⇒ *jvm-state type-error* ⇒ *jvm-state type-error* ⇒ *bool*
  (- ⊢ - −*jvmd*→ - [*61*,*61*,*61*]*60*) **where**
  exec-all-d-def1: *P* ⊢ σ −*jvmd*→ σ′ ⟷ (σ,σ′) ∈ (*exec-1-d P*)*

notation (*ASCII*)
  exec-all-d (- |− - −*jvmd*−> - [*61*,*61*,*61*]*60*)

lemma *exec-1-d-eq*:
exec-1-d P = {(*s*,*t*). ∃ σ. *s* = *Normal* σ ∧ *t* = *TypeError* ∧ *exec-d P* σ = *TypeError*} ∪
{(*s*,*t*). ∃ σ σ′. *s* = *Normal* σ ∧ *t* = *Normal* σ′ ∧ *exec-d P* σ = *Normal* (*Some* σ′)}
by (*auto elim*!: *exec-1-d.cases intro*!: *exec-1-d.intros*)

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

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

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

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

lemma *defensive-imp-aggressive*:
P ⊢ (*Normal* σ) −*jvmd*→ (*Normal* σ′) ⟹ *P* ⊢ σ −*jvm*→ σ′
end

3.7 Example for generating executable code from JVM semantics

theory *JVMListExample*
imports
../Common/SystemClasses
JVMExec
HOL−Library.Code-Target-Numeral
begin

definition *list-name* :: *string*
where
list-name == *''list''*

definition *test-name* :: *string*
where
test-name == *''test''*

definition *val-name* :: *string*
where
val-name == *''val''*

definition *next-name* :: *string*
where
next-name == *''next''*

definition *append-name* :: *string*
where
append-name == *''append''*

definition *makelist-name* :: *string*
where
makelist-name == *''makelist''*

definition *append-ins* :: *bytecode*
where
append-ins ==
[*Load 0*,
Getfield next-name list-name,
Load 0,
Getfield next-name list-name,
Push Null,
CmpEq,
IfFalse 7,
Pop,
Load 0,
Load 1,
Putfield next-name list-name,
Push Unit,
Return,
Load 1,
Invoke append-name 1,
Return]

definition *list-class* :: *jvm-method class*
where
list-class ==
(*Object*,
[(*val-name*, *Integer*), (*next-name*, *Class list-name*)],
[(*append-name*, [*Class list-name*], *Void*,
(*3*, *0*, *append-ins*, [(*1*, *2*, *NullPointer*, *7*, *0*)]))])

definition *make-list-ins* :: *bytecode*
where
make-list-ins ==
[*New list-name*,
Store 0,
Load 0,
Push (*Intg 1*),
Putfield val-name list-name,
New list-name,
Store 1,
Load 1,
Push (*Intg 2*),
Putfield val-name list-name,
New list-name,
Store 2,
Load 2,
Push (*Intg 3*),
Putfield val-name list-name,

```
    Load 0,
    Load 1,
    Invoke append-name 1,
    Pop,
    Load 0,
    Load 2,
    Invoke append-name 1,
    Return]
```

definition *test-class* :: *jvm-method class*
where
  test-class ==
   (*Object*, [],
    [(*makelist-name*, [], *Void*, (*3*, *2*, *make-list-ins*, []))])

definition *E* :: *jvm-prog*
where
  E == *SystemClasses* @ [(*list-name*, *list-class*), (*test-name*, *test-class*)]

definition *undefined-cname* :: *cname*
  where [*code del*]: *undefined-cname* = *undefined*
declare *undefined-cname-def*[*symmetric*, *code-unfold*]
code-printing constant *undefined-cname* ⇀ (*SML*) *object*

definition *undefined-val* :: *val*
  where [*code del*]: *undefined-val* = *undefined*
declare *undefined-val-def*[*symmetric*, *code-unfold*]
code-printing constant *undefined-val* ⇀ (*SML*) *Unit*

lemmas [*code-unfold*] = *SystemClasses-def* [*unfolded ObjectC-def NullPointerC-def ClassCastC-def OutOfMemoryC-def*]

definition *test* = *exec* (*E*, *start-state E test-name makelist-name*)

ML-val ‹
```
  @{code test};
  @{code exec} (@{code E}, @{code the} it);
  @{code exec} (@{code E}, @{code the} it);
  @{code exec} (@{code E}, @{code the} it);
  @{code exec} (@{code E}, @{code the} it);
  @{code exec} (@{code E}, @{code the} it);
  @{code exec} (@{code E}, @{code the} it);
  @{code exec} (@{code E}, @{code the} it);
  @{code exec} (@{code E}, @{code the} it);
  @{code exec} (@{code E}, @{code the} it);
  @{code exec} (@{code E}, @{code the} it);
  @{code exec} (@{code E}, @{code the} it);
  @{code exec} (@{code E}, @{code the} it);
  @{code exec} (@{code E}, @{code the} it);
  @{code exec} (@{code E}, @{code the} it);
  @{code exec} (@{code E}, @{code the} it);
  @{code exec} (@{code E}, @{code the} it);
  @{code exec} (@{code E}, @{code the} it);
  @{code exec} (@{code E}, @{code the} it);
```

```
@{code exec} (@{code E}, @{code the} it);
@{code exec} (@{code E}, @{code the} it);
@{code exec} (@{code E}, @{code the} it);
@{code exec} (@{code E}, @{code the} it);
@{code exec} (@{code E}, @{code the} it);
@{code exec} (@{code E}, @{code the} it);
@{code exec} (@{code E}, @{code the} it);
@{code exec} (@{code E}, @{code the} it);
@{code exec} (@{code E}, @{code the} it);
@{code exec} (@{code E}, @{code the} it);
@{code exec} (@{code E}, @{code the} it);
@{code exec} (@{code E}, @{code the} it);
@{code exec} (@{code E}, @{code the} it);
@{code exec} (@{code E}, @{code the} it);
@{code exec} (@{code E}, @{code the} it);
@{code exec} (@{code E}, @{code the} it);
@{code exec} (@{code E}, @{code the} it);
@{code exec} (@{code E}, @{code the} it);
@{code exec} (@{code E}, @{code the} it);
@{code exec} (@{code E}, @{code the} it);
@{code exec} (@{code E}, @{code the} it);
@{code exec} (@{code E}, @{code the} it);
@{code exec} (@{code E}, @{code the} it);
@{code exec} (@{code E}, @{code the} it);
@{code exec} (@{code E}, @{code the} it);
@{code exec} (@{code E}, @{code the} it);
@{code exec} (@{code E}, @{code the} it);
@{code exec} (@{code E}, @{code the} it);
@{code exec} (@{code E}, @{code the} it);
@{code exec} (@{code E}, @{code the} it);
@{code exec} (@{code E}, @{code the} it);
@{code exec} (@{code E}, @{code the} it);
@{code exec} (@{code E}, @{code the} it);
@{code exec} (@{code E}, @{code the} it);
@{code exec} (@{code E}, @{code the} it);
@{code exec} (@{code E}, @{code the} it);
@{code exec} (@{code E}, @{code the} it);
@{code exec} (@{code E}, @{code the} it);
@{code exec} (@{code E}, @{code the} it);
@{code exec} (@{code E}, @{code the} it);

val SOME (-, (h, -)) = it;
if snd (@{code the} (h (@{code nat-of-integer} 3))) (@{code val-name}, @{code list-name}) =
  SOME (@{code Intg} (@{code int-of-integer} 1)) then () else error wrong result;
if snd (@{code the} (h (@{code nat-of-integer} 3))) (@{code next-name}, @{code list-name}) =
  SOME (@{code Addr} (@{code nat-of-integer} 4)) then () else error wrong result;
if snd (@{code the} (h (@{code nat-of-integer} 4))) (@{code val-name}, @{code list-name}) =
  SOME (@{code Intg} (@{code int-of-integer} 2)) then () else error wrong result;
if snd (@{code the} (h (@{code nat-of-integer} 4))) (@{code next-name}, @{code list-name}) =
  SOME (@{code Addr} (@{code nat-of-integer} 5)) then () else error wrong result;
if snd (@{code the} (h (@{code nat-of-integer} 5))) (@{code val-name}, @{code list-name}) =
  SOME (@{code Intg} (@{code int-of-integer} 3)) then () else error wrong result;
if snd (@{code the} (h (@{code nat-of-integer} 5))) (@{code next-name}, @{code list-name}) =
  SOME @{code Null} then () else error wrong result;
```

76

›

end

Chapter 4

Bytecode Verifier

4.1 Semilattices

theory *Semilat*
imports *Main HOL−Library.While-Combinator*
begin

type-synonym *'a ord* = *'a* ⇒ *'a* ⇒ *bool*
type-synonym *'a binop* = *'a* ⇒ *'a* ⇒ *'a*
type-synonym *'a sl* = *'a set* × *'a ord* × *'a binop*

definition *lesub* :: *'a* ⇒ *'a ord* ⇒ *'a* ⇒ *bool*
where *lesub x r y* ⟷ *r x y*

definition *lesssub* :: *'a* ⇒ *'a ord* ⇒ *'a* ⇒ *bool*
where *lesssub x r y* ⟷ *lesub x r y* ∧ *x* ≠ *y*

definition *plussub* :: *'a* ⇒ (*'a* ⇒ *'b* ⇒ *'c*) ⇒ *'b* ⇒ *'c*
where *plussub x f y* = *f x y*

notation (*ASCII*)
lesub ((- /<='-- -) [*50*, *1000*, *51*] *50*) **and**
lesssub ((- /<'-- -) [*50*, *1000*, *51*] *50*) **and**
plussub ((- /+'-- -) [*65*, *1000*, *66*] *65*)

notation
lesub ((- /⊑_ -) [*50*, *0*, *51*] *50*) **and**
lesssub ((- /⊏_ -) [*50*, *0*, *51*] *50*) **and**
plussub ((- /⊔_ -) [*65*, *0*, *66*] *65*)

abbreviation (*input*)
lesub1 :: *'a* ⇒ *'a ord* ⇒ *'a* ⇒ *bool* ((- /⊑_ -) [*50*, *1000*, *51*] *50*)
where *x* ⊑*r* *y* == *x* ⊑*r* *y*

abbreviation (*input*)
lesssub1 :: *'a* ⇒ *'a ord* ⇒ *'a* ⇒ *bool* ((- /⊏_ -) [*50*, *1000*, *51*] *50*)
where *x* ⊏*r* *y* == *x* ⊏*r* *y*

abbreviation (*input*)

plussub1 :: $'a \Rightarrow ('a \Rightarrow 'b \Rightarrow 'c) \Rightarrow 'b \Rightarrow 'c$ ((- /$\sqcup$_ -) [65, 1000, 66] 65)
where $x \sqcup_f y == x \sqcup_f y$

definition *ord* :: $('a \times 'a)$ *set* $\Rightarrow$ $'a$ *ord*
where
ord r = $(\lambda x\ y.\ (x,y) \in r)$

definition *order* :: $'a$ *ord* $\Rightarrow$ $'a$ *set* $\Rightarrow$ *bool*
where
order r A $\longleftrightarrow$ $(\forall x{\in}A.\ x \sqsubseteq_r x) \wedge (\forall x{\in}A.\ \forall y{\in}A.\ x \sqsubseteq_r y \wedge y \sqsubseteq_r x \longrightarrow x{=}y) \wedge (\forall x{\in}A.\ \forall y{\in}A.\ \forall z{\in}A.\ x \sqsubseteq_r y \wedge y \sqsubseteq_r z \longrightarrow x \sqsubseteq_r z)$

definition *top* :: $'a$ *ord* $\Rightarrow$ $'a$ $\Rightarrow$ *bool*
where
top r T $\longleftrightarrow$ $(\forall x.\ x \sqsubseteq_r T)$

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

definition *closed* :: $'a$ *set* $\Rightarrow$ $'a$ *binop* $\Rightarrow$ *bool*
where
closed A f $\longleftrightarrow$ $(\forall x{\in}A.\ \forall y{\in}A.\ x \sqcup_f y \in A)$

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

definition *is-ub* :: $('a \times 'a)$ *set* $\Rightarrow$ $'a$ $\Rightarrow$ $'a$ $\Rightarrow$ $'a$ $\Rightarrow$ *bool*
where
is-ub r x y u $\longleftrightarrow$ $(x,u){\in}r \wedge (y,u){\in}r$

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

definition *some-lub* :: $('a \times 'a)$ *set* $\Rightarrow$ $'a$ $\Rightarrow$ $'a$ $\Rightarrow$ $'a$
where
some-lub r x y = $(SOME\ z.$ *is-lub r x y z*$)$

locale *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 A* $\Longrightarrow$ $x \in A \Longrightarrow x \sqsubseteq_r x$

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

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

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

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

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

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

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

lemma (**in** *Semilat*) *orderI* [*simp*, *intro*]: *order r A*

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*]: $x \in A \Longrightarrow x \sqsubseteq_r x$ **by** *auto*

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

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

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*:
assumes $x \in A$ **and** $y \in A$
shows $x \sqsubseteq_r y \longleftrightarrow x \sqcup_f y = y$ (**is** *?P* $\longleftrightarrow$ *?Q*)
lemma (**in** *Semilat*) *le-iff-plus-unchanged2*:
assumes $x \in A$ **and** $y \in A$
shows $x \sqsubseteq_r y \longleftrightarrow y \sqcup_f x = y$ (**is** *?P* $\longleftrightarrow$ *?Q*)
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*$\widehat{}$$*$) *x y y* = ((*x*,*y*)$\in$*r*$\widehat{}$$*$)
lemma *is-lub-bigger2* [*iff*]:
is-lub (*r*$\widehat{}$$*$) *x y x* = ((*y*,*x*)$\in$*r*$\widehat{}$$*$)
lemma *extend-lub*:
$[\![$ *single-valued r*; *is-lub* (*r*$\widehat{}$$*$) *x y u*; (*x*′,*x*) $\in$ *r* $]\!]$
$\Longrightarrow$ $\exists$ *v. is-lub* (*r*$\widehat{}$$*$) *x*′ *y v*
lemma *single-valued-has-lubs* [*rule-format*]:
$[\![$ *single-valued r*; (*x*,*u*) $\in$ *r*$\widehat{}$$*$ $]\!]$ $\Longrightarrow$ ($\forall$ *y.* (*y*,*u*) $\in$ *r*$\widehat{}$$*$ $\longrightarrow$
($\exists$ *z. is-lub* (*r*$\widehat{}$$*$) *x y z*))
lemma *some-lub-conv*:
$[\![$ *acyclic r*; *is-lub* (*r*$\widehat{}$$*$) *x y u* $]\!]$ $\Longrightarrow$ *some-lub* (*r*$\widehat{}$$*$) *x y* = *u*
lemma *is-lub-some-lub*:
$[\![$ *single-valued r*; *acyclic r*; (*x*,*u*)$\in$*r*$\widehat{}$$*$; (*y*,*u*)$\in$*r*$\widehat{}$$*$ $]\!]$
$\Longrightarrow$ *is-lub* (*r*$\widehat{}$$*$) *x y* (*some-lub* (*r*$\widehat{}$$*$) *x y*)

4.1.1 An executable lub-finder

definition *exec-lub* :: (′*a* $*$ ′*a*) *set* $\Rightarrow$ (′*a* $\Rightarrow$ ′*a*) $\Rightarrow$ ′*a binop*
where
exec-lub r f x y = *while* (λ*z.* (*x*,*z*) $\notin$ *r**) *f y*

lemma *exec-lub-refl*: *exec-lub r f T T* = *T*
by (*simp add*: *exec-lub-def while-unfold*)

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*$\widehat{}$$*$; (*y*,*u*):*r*$\widehat{}$$*$; $\forall$ *x y.* (*x*,*y*) $\in$ *r* $\longrightarrow$ *f x* = *y* $]\!]$
$\Longrightarrow$ *is-lub* (*r*$\widehat{}$$*$) *x y* (*exec-lub r f x y*)

end

4.2 The Error Type

theory *Err*
imports *Semilat*

begin

datatype *'a err* = *Err* | *OK 'a*

type-synonym *'a ebinop* = *'a* ⇒ *'a* ⇒ *'a err*
type-synonym *'a esl* = *'a set* × *'a ord* × *'a ebinop*

primrec *ok-val* :: *'a err* ⇒ *'a*
where
ok-val (*OK x*) = *x*

definition *lift* :: (*'a* ⇒ *'b err*) ⇒ (*'a err* ⇒ *'b err*)
where
lift f e = (*case e of Err* ⇒ *Err* | *OK x* ⇒ *f x*)

definition *lift2* :: (*'a* ⇒ *'b* ⇒ *'c err*) ⇒ *'a err* ⇒ *'b err* ⇒ *'c err*
where
lift2 f e_1 e_2 =
(*case* e_1 *of Err* ⇒ *Err* | *OK x* ⇒ (*case* e_2 *of Err* ⇒ *Err* | *OK y* ⇒ *f x y*))

definition *le* :: *'a ord* ⇒ *'a err ord*
where
le r e_1 e_2 =
(*case* e_2 *of Err* ⇒ *True* | *OK y* ⇒ (*case* e_1 *of Err* ⇒ *False* | *OK x* ⇒ *x* ⊑$_r$ *y*))

definition *sup* :: (*'a* ⇒ *'b* ⇒ *'c*) ⇒ (*'a err* ⇒ *'b err* ⇒ *'c err*)
where
sup f = *lift2* (λ*x y*. *OK* (*x* ⊔$_f$ *y*))

definition *err* :: *'a set* ⇒ *'a err set*
where
err A = *insert Err* {*OK x*|*x*. *x*∈*A*}

definition *esl* :: *'a sl* ⇒ *'a esl*
where
esl = (λ(*A*,*r*,*f*). (*A*, *r*, λ*x y*. *OK*(*f x y*)))

definition *sl* :: *'a esl* ⇒ *'a err sl*
where
sl = (λ(*A*,*r*,*f*). (*err A*, *le r*, *lift2 f*))

abbreviation
err-semilat :: *'a esl* ⇒ *bool* **where**
err-semilat L == *semilat*(*sl L*)

primrec *strict* :: (*'a* ⇒ *'b err*) ⇒ (*'a err* ⇒ *'b err*)
where
strict f Err = *Err*
| *strict f* (*OK x*) = *f x*

lemma *err-def'*:
err A = *insert Err* {*x*. ∃ *y*∈*A*. *x* = *OK y*}
lemma *strict-Some* [*simp*]:
(*strict f x* = *OK y*) = (∃ *z*. *x* = *OK z* ∧ *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*: $e1 \sqsubseteq_{le\ r} e2 = le\ r\ e1\ e2$
lemma *le-err-refl*: $\forall\, x.\ x \sqsubseteq_r x \Longrightarrow e \sqsubseteq_{le\ r} e$
lemma *le-err-refl'*: $(\forall\, x{\in}A.\ x \sqsubseteq_r x) \Longrightarrow e \in err\ A \Longrightarrow\ e \sqsubseteq_{le\ r} e$

4.3 More about Options

theory *Opt* **imports** *Err* **begin**

definition *le* :: *'a ord* ⇒ *'a option ord*
where
le r o_1 o_2 =
(*case* o_2 *of None* ⇒ o_1=*None* | *Some y* ⇒ (*case* o_1 *of None* ⇒ *True* | *Some x* ⇒ $x \sqsubseteq_r y$))

definition *opt* :: *'a set* ⇒ *'a option set*
where
opt A = *insert None* {*Some y* |*y*. $y \in A$}

definition *sup* :: *'a ebinop* ⇒ *'a option ebinop*
where
sup f o_1 o_2 =
(*case* o_1 *of None* ⇒ *OK* o_2
| *Some x* ⇒ (*case* o_2 *of None* ⇒ *OK* o_1
| *Some y* ⇒ (*case f x y of Err* ⇒ *Err* | *OK z* ⇒ *OK* (*Some z*))))

definition *esl* :: *'a esl* ⇒ *'a option esl*
where
esl = (λ(*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* ⇒ o_1=*None* |
Some y ⇒ (*case* o_1 *of None* ⇒ *True* | *Some x* ⇒ $x \sqsubseteq_r y$))
lemma *le-opt-refl*: *order r A* $\Longrightarrow\ x \in opt\ A \Longrightarrow x \sqsubseteq_{le\ r} x$

4.4 Products as Semilattices

theory *Product*
imports *Err*
begin

definition *le* :: *'a ord* ⇒ *'b ord* ⇒ (*'a* × *'b*) *ord*
where
le r_A r_B = (λ(a_1,b_1) (a_2,b_2). $a_1 \sqsubseteq_{r_A} a_2 \wedge b_1 \sqsubseteq_{r_B} b_2$)

definition *sup* :: *'a ebinop* ⇒ *'b ebinop* ⇒ (*'a* × *'b*) *ebinop*
where
sup f g = (λ(a_1,b_1)(a_2,b_2). *Err.sup Pair* ($a_1 \sqcup_f a_2$) ($b_1 \sqcup_g b_2$))

definition *esl* :: *'a esl* ⇒ *'b esl* ⇒ (*'a* × *'b*) *esl*
where

esl = (λ(*A*,r_A,f_A) (*B*,r_B,f_B). (*A* × *B*, *le* r_A r_B, *sup* f_A f_B))

abbreviation
lesubprod :: 'a × 'b ⇒ ('a ⇒ 'a ⇒ *bool*) ⇒ ('b ⇒ 'b ⇒ *bool*) ⇒ 'a × 'b ⇒ *bool*
((- /⊑'(-,-') -) [*50*, *0*, *0*, *51*] *50*) **where**
p ⊑(*rA*,*rB*) *q* == *p* ⊑$_{Product.le\ rA\ rB}$ *q*

lemma *unfold-lesub-prod*: *x* ⊑(r_A,r_B) *y* = *le* r_A r_B *x* *y*
lemma *le-prod-Pair-conv* [*iff*]: ((*a*$_1$,*b*$_1$) ⊑(r_A,r_B) (*a*$_2$,*b*$_2$)) = (*a*$_1$ ⊑$_{r_A}$ *a*$_2$ & *b*$_1$ ⊑$_{r_B}$ *b*$_2$)
lemma *less-prod-Pair-conv*:
((*a*$_1$,*b*$_1$) ⊏$_{Product.le\ r_A\ r_B}$ (*a*$_2$,*b*$_2$)) =
(*a*$_1$ ⊏$_{r_A}$ *a*$_2$ & *b*$_1$ ⊑$_{r_B}$ *b*$_2$ | *a*$_1$ ⊑$_{r_A}$ *a*$_2$ & *b*$_1$ ⊏$_{r_B}$ *b*$_2$)
lemma *order-le-prodI* [*iff*]: (*order* r_A *A* & *order* r_B *B*) ⟹ *order* (*Product.le* r_A r_B) (*A* × *B*)
apply (*unfold order-def*)
apply *safe*
apply *blast*+
done

lemma *order-le-prodE*: *A* ≠ {} ⟹ *B* ≠ {} ⟹ *order* (*Product.le* r_A r_B) (*A* × *B*) ⟹ (*order* r_A *A* & *order* r_B *B*)
apply (*unfold order-def*)
apply *simp*
apply *safe*
apply *blast*+
done

lemma *order-le-prod* [*iff*]: *A* ≠ {} ⟹ *B* ≠ {} ⟹ *order*(*Product.le* r_A r_B) (*A* × *B*) = (*order* r_A *A* & *order* r_B *B*)
lemma *acc-le-prodI* [*intro*!]:
⟦ *acc* r_A; *acc* r_B ⟧ ⟹ *acc*(*Product.le* r_A r_B)

lemma *closed-lift2-sup*:
⟦ *closed* (*err A*) (*lift2 f*); *closed* (*err B*) (*lift2 g*) ⟧ ⟹
closed (*err*(*A*×*B*)) (*lift2*(*sup f g*))
lemma *unfold-plussub-lift2*: *e*$_1$ ⊔$_{lift2\ f}$ *e*$_2$ = *lift2 f* *e*$_1$ *e*$_2$

lemma *plus-eq-Err-conv* [*simp*]:
assumes *x*∈*A* *y*∈*A* *semilat*(*err A*, *Err.le r*, *lift2 f*)
shows (*x* ⊔$_f$ *y* = *Err*) = (¬(∃ *z*∈*A*. *x* ⊑$_r$ *z* ∧ *y* ⊑$_r$ *z*))
lemma *err-semilat-Product-esl*:
⋀*L*$_1$ *L*$_2$. ⟦ *err-semilat L*$_1$; *err-semilat L*$_2$ ⟧ ⟹ *err-semilat*(*Product.esl L*$_1$ *L*$_2$)
end

4.5 Fixed Length Lists

theory *Listn*
imports *Err HOL−Library.NList*
begin

definition *le* :: 'a *ord* ⇒ ('a *list*)*ord*
where
le r = *list-all2* (λ*x* *y*. *x* ⊑$_r$ *y*)

abbreviation
lesublist :: *'a list* ⇒ *'a ord* ⇒ *'a list* ⇒ *bool* ((- /[⊑_] -) [*50*, *0*, *51*] *50*) **where**
x [⊑*r*] *y* == *x* <=-(*Listn.le r*) *y*

abbreviation
lesssublist :: *'a list* ⇒ *'a ord* ⇒ *'a list* ⇒ *bool* ((- /[⊏_] -) [*50*, *0*, *51*] *50*) **where**
x [⊏*r*] *y* == *x* <-(*Listn.le r*) *y*

abbreviation
plussublist :: *'a list* ⇒ (*'a* ⇒ *'b* ⇒ *'c*) ⇒ *'b list* ⇒ *'c list*
((- /[⊔_] -) [*65*, *0*, *66*] *65*) **where**
x [⊔*f*] *y* == *x* ⊔*map2 f y*

primrec *coalesce* :: *'a err list* ⇒ *'a list err*
where
coalesce [] = *OK*[]
| *coalesce* (*ex*#*exs*) = *Err.sup* (#) *ex* (*coalesce exs*)

definition *sl* :: *nat* ⇒ *'a sl* ⇒ *'a list sl*
where
sl n = (λ(*A*,*r*,*f*). (*nlists n A*, *le r*, *map2 f*))

definition *sup* :: (*'a* ⇒ *'b* ⇒ *'c err*) ⇒ *'a list* ⇒ *'b list* ⇒ *'c list err*
where
sup f = (λ*xs ys*. *if size xs* = *size ys then coalesce*(*xs* [⊔*f*] *ys*) *else Err*)

definition *upto-esl* :: *nat* ⇒ *'a esl* ⇒ *'a list esl*
where
upto-esl m = (λ(*A*,*r*,*f*). (*Union*{*nlists n A* |*n*. *n* ≤ *m*}, *le r*, *sup f*))

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

lemma *unfold-lesub-list*: *xs* [⊑*r*] *ys* = *Listn.le r xs ys*
lemma *Nil-le-conv* [*iff*]: ([] [⊑*r*] *ys*) = (*ys* = [])
lemma *Cons-notle-Nil* [*iff*]: ¬ *x*#*xs* [⊑*r*] []
lemma *Cons-le-Cons* [*iff*]: *x*#*xs* [⊑*r*] *y*#*ys* = (*x* ⊑*r* *y* ∧ *xs* [⊑*r*] *ys*)
lemma *list-update-le-cong*:
⟦ *i*<*size xs*; *xs* [⊑*r*] *ys*; *x* ⊑*r* *y* ⟧ ⟹ *xs*[*i*:=*x*] [⊑*r*] *ys*[*i*:=*y*]

lemma *le-listD*: ⟦ *xs* [⊑*r*] *ys*; *p* < *size xs* ⟧ ⟹ *xs*!*p* ⊑*r* *ys*!*p*
lemma *le-list-refl*: ∀ *x*. *x* ⊑*r* *x* ⟹ *xs* [⊑*r*] *xs*

lemma *le-list-trans*:
assumes *ord*: *order r A*
and *xs*: *xs* ∈ *nlists n A* **and** *ys*: *ys* ∈ *nlists n A* **and** *zs*: *zs* ∈ *nlists n A*
and *xs* [⊑*r*] *ys* **and** *ys* [⊑*r*] *zs*
shows *xs* [⊑*r*] *zs*

4.6 Typing and Dataflow Analysis Framework

theory *Typing-Framework-1* **imports** *Semilattices* **begin**

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

type-synonym
's step-type = *nat* ⇒ *'s* ⇒ (*nat* × *'s*) *list*

definition *stable* :: *'s ord* ⇒ *'s step-type* ⇒ *'s list* ⇒ *nat* ⇒ *bool*
where
stable r step τs p ⟷ (∀ (*q*,*τ*) ∈ *set* (*step p* (*τs*!*p*)). *τ* ⊑$_r$ *τs*!*q*)

definition *stables* :: *'s ord* ⇒ *'s step-type* ⇒ *'s list* ⇒ *bool*
where
stables r step τs ⟷ (∀ *p* < *size τs*. *stable r step τs p*)

definition *wt-step* :: *'s ord* ⇒ *'s* ⇒ *'s step-type* ⇒ *'s list* ⇒ *bool*
where
wt-step r T step τs ⟷ (∀ *p*<*size τs*. *τs*!*p* ≠ *T* ∧ *stable r step τs p*)

end

4.7 More on Semilattices

theory *SemilatAlg*
imports *Typing-Framework-1*
begin

definition *lesubstep-type* :: (*nat* × *'s*) *set* ⇒ *'s ord* ⇒ (*nat* × *'s*) *set* ⇒ *bool*
((- /{⊑$_-$} -) [*50*, *0*, *51*] *50*)
where *A* {⊑$_r$} *B* ≡ ∀ (*p*,*τ*) ∈ *A*. ∃ *τ'*. (*p*,*τ'*) ∈ *B* ∧ *τ* ⊑$_r$ *τ'*

notation (*ASCII*)
lesubstep-type ((- /{<='--} -) [*50*, *0*, *51*] *50*)

primrec *pluslussub* :: *'a list* ⇒ (*'a* ⇒ *'a* ⇒ *'a*) ⇒ *'a* ⇒ *'a* ((- /⨆$_-$ -) [*65*, *0*, *66*] *65*)
where
pluslussub [] *f y* = *y*
| *pluslussub* (*x*#*xs*) *f y* = *pluslussub xs f* (*x* ⊔$_f$ *y*)
definition *bounded* :: *'s step-type* ⇒ *nat* ⇒ *bool*
where
bounded step n ⟷ (∀ *p*<*n*. ∀ *τ*. ∀ (*q*,*τ'*) ∈ *set* (*step p τ*). *q*<*n*)

definition *pres-type* :: *'s step-type* ⇒ *nat* ⇒ *'s set* ⇒ *bool*
where
pres-type step n A ⟷ (∀ *τ*∈*A*. ∀ *p*<*n*. ∀ (*q*,*τ'*) ∈ *set* (*step p τ*). *τ'* ∈ *A*)

definition *mono* :: *'s ord* ⇒ *'s step-type* ⇒ *nat* ⇒ *'s set* ⇒ *bool*
where
mono r step n A ⟷
(∀ *τ p τ'*. *τ* ∈ *A* ∧ *p*<*n* ∧ *τ* ⊑$_r$ *τ'* ⟶ *set* (*step p τ*) {⊑$_r$} *set* (*step p τ'*))

lemma [*iff*]: {} {⊑$_r$} *B*

lemma [*iff*]: (*A* {⊑$_r$} {}) = (*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-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*:
$A\ \{\sqsubseteq_r\}\ B \Longrightarrow (x,y) \in A \Longrightarrow \exists\, y'.\ (x,\ y') \in B \wedge y \sqsubseteq_r y'$

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*: **assumes** *Semilat A r f* **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*: $z \in A$
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'*: **assumes** *Semilat A r f*
shows $[\![\forall (p,s) \in set\ S.\ s \in A;\ y \in A;\ (a,b) \in set\ S]\!]$
$\Longrightarrow b \sqsubseteq_r map\ snd\ [(p',\ t') \leftarrow S.\ p' = a] \bigsqcup_f y$

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

end

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

theory *Typing-Framework-err* **imports** *SemilatAlg* **begin**

definition *wt-err-step* :: *'s ord* ⇒ *'s err step-type* ⇒ *'s err list* ⇒ *bool*
where
wt-err-step r step τs ⟷ *wt-step* (*Err.le r*) *Err step τs*

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

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

definition *error* :: $nat \Rightarrow (nat \times 'a\ err)\ list$
where
$error\ n = map\ (\lambda x.\ (x,Err))\ [0..<n]$

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

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

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*:
$[\![\ order\ r\ A;\ app\text{-}mono\ r\ app\ n\ A;\ bounded\ (err\text{-}step\ n\ app\ step)\ n;$
$\quad \forall s\ p\ t.\ s \in A \wedge p < n \wedge s \sqsubseteq_r t \longrightarrow app\ p\ t \longrightarrow set\ (step\ p\ s)\ \{\sqsubseteq_r\}\ set\ (step\ p\ t)\ ]\!]$
$\Longrightarrow\ mono\ (Err.le\ r)\ (err\text{-}step\ n\ app\ step)\ n\ (err\ A)$
lemma *in-errorD*: $(x,y) \in set\ (error\ n) \Longrightarrow y = Err$
lemma *pres-type-lift*:
$\forall s{\in}A.\ \forall p.\ p < n \longrightarrow app\ p\ s \longrightarrow (\forall (q,\ s'){\in}set\ (step\ p\ s).\ s' \in A)$
$\Longrightarrow pres\text{-}type\ (err\text{-}step\ n\ app\ step)\ n\ (err\ A)$

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-1*
imports *SemilatAlg*
begin

primrec *merges* :: *'s binop* ⇒ (*nat* × *'s*) *list* ⇒ *'s list* ⇒ *'s list*
where
merges f [] *τs* = *τs*
| *merges f* (*p'*#*ps*) *τs* = (*let* (*p*,*τ*) = *p' in merges f ps* (*τs*[*p* := *τ* ⊔*f τs*!*p*]))

lemmas [*simp*] = *Let-def Semilat.le-iff-plus-unchanged* [*OF Semilat.intro*, *symmetric*]

lemma (**in** *Semilat*) *nth-merges*:
⋀*ss*. ⟦*p* < *length ss*; *ss* ∈ *nlists n A*; ∀(*p*,*t*)∈*set ps*. *p*<*n* ∧ *t*∈*A* ⟧ ⟹
(*merges f ps ss*)!*p* = *map snd* [(*p'*,*t'*) ← *ps*. *p'*=*p*] ⨆*f ss*!*p*
(**is** ⋀*ss*. ⟦-; -; *?steptype ps*⟧ ⟹ *?P ss ps*)

lemma *length-merges* [*simp*]:
⋀*ss*. *size*(*merges f ps ss*) = *size ss*
lemma (**in** *Semilat*) *merges-preserves-type-lemma*:
shows ∀*xs*. *xs* ∈ *nlists n A* ⟶ (∀(*p*,*x*) ∈ *set ps*. *p*<*n* ∧ *x*∈*A*)
⟶ *merges f ps xs* ∈ *nlists n A*
lemma (**in** *Semilat*) *merges-preserves-type* [*simp*]:
⟦ *xs* ∈ *nlists n A*; ∀(*p*,*x*) ∈ *set ps*. *p*<*n* ∧ *x*∈*A* ⟧
⟹ *merges f ps xs* ∈ *nlists n A*
by (*simp add*: *merges-preserves-type-lemma*)

lemma (**in** *Semilat*) *list-update-le-listI* [*rule-format*]:
set xs ⊆ *A* ⟶ *set ys* ⊆ *A* ⟶ *xs* [⊑*r*] *ys* ⟶ *p* < *size xs* ⟶
x ⊑*r ys*!*p* ⟶ *x*∈*A* ⟶ *xs*[*p* := *x* ⊔*f xs*!*p*] [⊑*r*] *ys*
lemma (**in** *Semilat*) *merges-pres-le-ub*:
assumes *set ts* ⊆ *A* *set ss* ⊆ *A*
∀(*p*,*t*)∈*set ps*. *t* ⊑*r ts*!*p* ∧ *t* ∈ *A* ∧ *p* < *size ts* *ss* [⊑*r*] *ts*
shows *merges f ps ss* [⊑*r*] *ts*

end

4.10 Kildall's Algorithm

theory *Kildall-2*
imports *SemilatAlg Kildall-1*
begin

primrec *propa* :: *'s binop* ⇒ *(nat* × *'s) list* ⇒ *'s list* ⇒ *nat set* ⇒ *'s list* ∗ *nat set*
where
propa f [] *τs w* = *(τs,w)*
| *propa f (q'#qs) τs w* = *(let (q,τ)* = *q'*;
u = *τ* ⊔*f τs!q*;
w' = *(if u* = *τs!q then w else insert q w)*
in propa f qs (τs[q := *u]) w')*

definition *iter* :: *'s binop* ⇒ *'s step-type* ⇒
's list ⇒ *nat set* ⇒ *'s list* × *nat set*
where
iter f step τs w =
while (λ*(τs,w). w* ≠ {})
(λ*(τs,w). let p* = *SOME p. p* ∈ *w*
in propa f (step p (τs!p)) τs (w−{*p*}))
(τs,w)

definition *unstables* :: *'s ord* ⇒ *'s step-type* ⇒ *'s list* ⇒ *nat set*
where
unstables r step τs = {*p. p* < *size τs* ∧ ¬*stable r step τs p*}

definition *kildall* :: *'s ord* ⇒ *'s binop* ⇒ *'s step-type* ⇒ *'s list* ⇒ *'s list*
where
kildall r f step τs = *fst*(*iter f step τs* (*unstables r step τs*))

lemma (**in** *Semilat*) *merges-incr-lemma*:
∀ *xs. xs* ∈ *nlists n A* ⟶ (∀ *(p,x)*∈*set ps. p*<*size xs* ∧ *x* ∈ *A*) ⟶ *xs* [⊑*r*] *merges f ps xs*
apply (*induct ps*)
apply *simp*
apply(*insert orderI*)
apply (*fastforce intro*:*le-list-refl'*)
apply *simp*
apply *clarify*
apply (*rule order-trans*)
defer
apply (*erule list-update-incr*)
apply *simp*+
done

lemma (**in** *Semilat*) *merges-incr*:
⟦ *xs* ∈ *nlists n A*; ∀ *(p,x)*∈*set ps. p*<*size xs* ∧ *x* ∈ *A* ⟧
⟹ *xs* [⊑*r*] *merges f ps xs*

by (*simp add*: *merges-incr-lemma*)

lemma (**in** *Semilat*) *merges-same-conv* [*rule-format*]:
$(\forall\, xs.\ xs \in nlists\ n\ A \longrightarrow (\forall\,(p,x)\in set\ ps.\ p<size\ xs \wedge x\in A) \longrightarrow$
$(merges\ f\ ps\ xs = xs) = (\forall\,(p,x)\in set\ ps.\ x \sqsubseteq_r xs!p))$
lemma *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 $[\![pres\text{-}type\ step\ n\ A;\ bounded\ step\ n;$
$ss \in nlists\ n\ A;\ p \in w;\ \forall\, q\in w.\ q < n;$
$\forall\, q.\ q < n \longrightarrow q \notin w \longrightarrow stable\ r\ step\ ss\ q;\ q < n;$
$\forall\, s'.\ (q,s') \in set\ (step\ p\ (ss!p)) \longrightarrow s' \sqcup_f ss!q = ss!q;$
$q \notin w \vee q = p\ ]\!]$
$\Longrightarrow stable\ r\ step\ (merges\ f\ (step\ p\ (ss!p))\ ss)\ q$

lemma (**in** *Semilat*) *merges-bounded-lemma*:
$[\![\ mono\ r\ step\ n\ A;\ bounded\ step\ n;\ pres\text{-}type\ step\ n\ A;$
$\forall\,(p',s') \in set\ (step\ p\ (ss!p)).\ s' \in A;\ ss \in nlists\ n\ A;\ ts \in nlists\ n\ A;\ p < n;$
$ss\ [\sqsubseteq_r]\ ts;\ \forall\, p.\ p < n \longrightarrow stable\ r\ step\ ts\ p\ ]\!]$
$\Longrightarrow merges\ f\ (step\ p\ (ss!p))\ ss\ [\sqsubseteq_r]\ ts$

lemma *termination-lemma*: **assumes** *Semilat A r f*
shows $[\![\ ss \in nlists\ n\ A;\ \forall\,(q,t)\in set\ qs.\ q<n \wedge t\in A;\ p\in w\ ]\!] \Longrightarrow$
$ss\ [\sqsubset_r]\ merges\ f\ qs\ ss \vee$
$merges\ f\ qs\ ss = ss \wedge \{q.\ \exists\, t.\ (q,t)\in set\ qs \wedge t \sqcup_f ss!q \neq ss!q\} \cup (w-\{p\}) \subset w$
end

4.11 The Lightweight Bytecode Verifier

theory *LBVSpec*
imports *SemilatAlg Opt*
begin

type-synonym
$'s\ certificate = 's\ list$

primrec $merge :: 's\ certificate \Rightarrow 's\ binop \Rightarrow 's\ ord \Rightarrow 's \Rightarrow nat \Rightarrow (nat \times 's)\ list \Rightarrow 's \Rightarrow 's$
where
$merge\ cert\ f\ r\ T\ pc\ []\ \ \ x = x$
$|\ merge\ cert\ f\ r\ T\ pc\ (s\#ss)\ x = merge\ cert\ f\ r\ T\ pc\ ss\ (let\ (pc',s') = s\ in$
$\quad if\ pc'=pc+1\ then\ s' \sqcup_f x$
$\quad else\ if\ s' \sqsubseteq_r cert!pc'\ then\ x$
$\quad else\ T)$

definition $wtl\text{-}inst :: 's\ certificate \Rightarrow 's\ binop \Rightarrow 's\ ord \Rightarrow 's \Rightarrow$
$'s\ step\text{-}type \Rightarrow nat \Rightarrow 's \Rightarrow 's$
where
$wtl\text{-}inst\ cert\ f\ r\ T\ step\ pc\ s = merge\ cert\ f\ r\ T\ pc\ (step\ pc\ s)\ (cert!(pc+1))$

definition *wtl-cert* :: *'s certificate* ⇒ *'s binop* ⇒ *'s ord* ⇒ *'s* ⇒ *'s* ⇒
's step-type ⇒ *nat* ⇒ *'s* ⇒ *'s*
where
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*)

primrec *wtl-inst-list* :: *'a list* ⇒ *'s certificate* ⇒ *'s binop* ⇒ *'s ord* ⇒ *'s* ⇒ *'s* ⇒
's step-type ⇒ *nat* ⇒ *'s* ⇒ *'s*
where
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'*)

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

definition *bottom* :: *'a ord* ⇒ *'a* ⇒ *bool*
where
bottom r B ⟷ (∀ *x*. *B* ⊑$_r$ *x*)

locale *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*: ⊥ ∈ *A*

fixes *merge* :: *'a certificate* ⇒ *nat* ⇒ (*nat* × *'a*) *list* ⇒ *'a* ⇒ *'a*
defines *mrg-def*: *merge cert* ≡ *LBVSpec.merge cert f r* ⊤

fixes *wti* :: *'a certificate* ⇒ *nat* ⇒ *'a* ⇒ *'a*
defines *wti-def*: *wti cert* ≡ *wtl-inst cert f r* ⊤ *step*

fixes *wtc* :: *'a certificate* ⇒ *nat* ⇒ *'a* ⇒ *'a*
defines *wtc-def*: *wtc cert* ≡ *wtl-cert cert f r* ⊤ ⊥ *step*

fixes *wtl* :: *'b list* ⇒ *'a certificate* ⇒ *nat* ⇒ *'a* ⇒ *'a*
defines *wtl-def*: *wtl ins cert* ≡ *wtl-inst-list ins cert f r* ⊤ ⊥ *step*

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

lemma (**in** *lbv*) *wtc*:
wtc c pc s = (*if c*!*pc* = ⊥ *then wti c pc s else if s* ⊑$_r$ *c*!*pc then wti c pc* (*c*!*pc*) *else* ⊤)

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*]

4.11.1 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'*) $\leftarrow$ *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* $\in$ *A* $\Longrightarrow$ *x* $\sqsubseteq_r$ $\bot$ $=$ (*x* $=$ $\bot$)
using *B-A* **by** (*blast intro*: *antisym-r*)

4.11.2 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 $[(p',t') \leftarrow ss.\ p'=pc+1] \bigsqcup_f x$
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* $[(p',t') \leftarrow ss.\ p'=pc+1] \bigsqcup_f x$)

4.11.3 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) ≠ ⊤*

4.11.4 preserves-type

lemma (**in** *lbv*) *merge-pres*:
assumes *s0*: *snd'set ss ⊆ A* **and** *x*: *x ∈ A*
shows *merge c pc ss x ∈ A*
lemma *pres-typeD2*:
pres-type step n A ⟹ s ∈ A ⟹ p < n ⟹ snd'set (step p s) ⊆ A
by *auto* (*drule pres-typeD*)

lemma (**in** *lbv*) *wti-pres* [*intro?*]:
assumes *pres*: *pres-type step n A*
assumes *cert*: *c!(pc+1) ∈ A*
assumes *s-pc*: *s ∈ A pc < n*
shows *wti c pc s ∈ A*
lemma (**in** *lbv*) *wtc-pres*:
assumes *pres-type step n A*
assumes *c!pc ∈ A* **and** *c!(pc+1) ∈ A*
assumes *s ∈ A* **and** *pc < n*
shows *wtc c pc s ∈ A*
lemma (**in** *lbv*) *wtl-pres*:
assumes *pres*: *pres-type step (length is) A*
assumes *cert*: *cert-ok c (length is) ⊤ ⊥ A*
assumes *s*: *s ∈ A*
assumes *all*: *wtl is c 0 s ≠ ⊤*
shows *pc < length is ⟹ wtl (take pc is) c 0 s ∈ A*
(**is** *?len pc ⟹ ?wtl pc ∈ A*)
end

4.12 Correctness of the LBV

theory *LBVCorrect*
imports *LBVSpec Typing-Framework-1*
begin

locale *lbvs = lbv +*
fixes s_0 :: *'a*
fixes *c* :: *'a list*
fixes *ins* :: *'b list*
fixes *τs* :: *'a list*
defines *phi-def*:
τs ≡ map (λpc. if c!pc = ⊥ 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) ⊤ ⊥ A*
assumes *pres*: *pres-type step (size ins) A*

lemma (**in** *lbvs*) *phi-None* [*intro?*]:
⟦ pc < size ins; c!pc = ⊥ ⟧ ⟹ τs!pc = wtl (take pc ins) c 0 s_0
lemma (**in** *lbvs*) *phi-Some* [*intro?*]:
⟦ pc < size ins; c!pc ≠ ⊥ ⟧ ⟹ τs!pc = c!pc
lemma (**in** *lbvs*) *phi-len* [*simp*]: *size τs = size ins*

lemma (**in** *lbvs*) *wtl-suc-pc*:
assumes *all*: *wtl ins c 0* $s_0 \neq \top$
assumes *pc*: *pc+1* < *size ins*
assumes *sA*: $s_0 \in A$
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 $\tau s \in$ *nlists* (*size ins*) *A*
lemma (**in** *lbvs*) *phi0*:
assumes *wtl*: *wtl ins c 0* $s_0 \neq \top$ **and** *0*: *0* < *size ins* **and** s_0: $s_0 \in A$
shows $s_0 \sqsubseteq_r \tau s!0$

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

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

4.13 Completeness of the LBV

theory *LBVComplete*
imports *LBVSpec Typing-Framework-1*
begin

definition *is-target* :: *'s step-type* ⇒ *'s list* ⇒ *nat* ⇒ *bool* **where**
is-target step τs *pc'* ⟷ (∃ *pc s'*. *pc'* ≠ *pc+1* ∧ *pc* < *size* τs ∧ (*pc'*,*s'*) ∈ *set* (*step pc* ($\tau s!pc$)))

definition *make-cert* :: *'s step-type* ⇒ *'s list* ⇒ *'s* ⇒ *'s certificate* **where**
make-cert step τs *B* = *map* (λ*pc*. *if is-target step* τs *pc then* $\tau s!pc$ *else B*) [*0*..<*size* τs] @ [*B*]

lemma [*code*]:
is-target step τs *pc'* =
list-ex (λ*pc*. *pc'* ≠ *pc+1* ∧ *List.member* (*map fst* (*step pc* ($\tau s!pc$))) *pc'*) [*0*..<*size* τs]
locale *lbvc* = *lbv* +
fixes τs :: *'a list*
fixes *c* :: *'a list*
defines *cert-def*: *c* ≡ *make-cert step* τs ⊥

assumes *mono*: *mono r step* (*size* τs) *A*
assumes *pres*: *pres-type step* (*size* τs) *A*
assumes τs: ∀ *pc* < *size* τs. $\tau s!pc \in A \wedge \tau s!pc \neq \top$
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'*) $\in$ *set* (*step pc* (τs!*pc*));
pc' $\neq$ *pc+1*; *pc* < *size* τs; *pc'* < *size* τs $]\!]$
$\Longrightarrow$ *c*!*pc'* = τs!*pc'*
lemma (**in** *lbvc*) *cert-approx* [*intro?*]:
$[\![$ *pc* < *size* τs; *c*!*pc* $\neq \bot$ $]\!] \Longrightarrow$ *c*!*pc* = τs!*pc*
lemma (**in** *lbv*) *le-top* [*simp*, *intro*]: *x* <=*-r* $\top$
lemma (**in** *lbv*) *merge-mono*:
assumes *less*: *set* ss_2 {$\sqsubseteq_r$} *set* ss_1
assumes *x*: *x* $\in$ *A*
assumes ss_1: *snd*'*set* ss_1 $\subseteq$ *A*
assumes ss_2: *snd*'*set* ss_2 $\subseteq$ *A*
assumes *boun*: $\forall x \in$(*fst*'*set* ss_1). *x* < *size* τs
assumes *cert*: *cert-ok c* (*size* τs) *T B A*
shows *merge c pc* ss_2 *x* $\sqsubseteq_r$ *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* τs **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* τs **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*]: $x \in A \Longrightarrow \top \sqsubseteq_r x = (x = \top)$
lemma (**in** *lbv*) *neq-top* [*simp*, *elim*]: $[\![x \sqsubseteq_r y; y \neq \top; y \in A]\!] \Longrightarrow x \neq \top$
lemma (**in** *lbvc*) *stable-wti*:
assumes *stable*: *stable r step* τs *pc* **and** *pc*: *pc* < *size* τs
shows *wti c pc* (τs!*pc*) $\neq \top$
lemma (**in** *lbvc*) *wti-less*:
assumes *stable*: *stable r step* τs *pc* **and** *suc-pc*: *Suc pc* < *size* τs
shows *wti c pc* (τs!*pc*) $\sqsubseteq_r$ τs!*Suc pc* (**is** *?wti* $\sqsubseteq_r$ -)
lemma (**in** *lbvc*) *stable-wtc*:
assumes *stable*: *stable r step* τs *pc* **and** *pc*: *pc* < *size* τs
shows *wtc c pc* (τs!*pc*) $\neq \top$
lemma (**in** *lbvc*) *wtc-less*:
assumes *stable*: *stable r step* τs *pc* **and** *suc-pc*: *Suc pc* < *size* τs
shows *wtc c pc* (τs!*pc*) $\sqsubseteq_r$ τs!*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* τs $\Longrightarrow$ *s* $\sqsubseteq_r$ τs!*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*: *wt-step r* $\top$ *step* τs
assumes *s*: *s* $\sqsubseteq_r$ τs!*0 s* $\in$ *A s* $\neq \top$ **and** *eq*: *size ins* = *size* τs
shows *wtl ins c 0 s* $\neq \top$
end

4.14 The Jinja Type System as a Semilattice

theory *SemiType*
imports *../Common/WellForm ../DFA/Semilattices*
begin

definition *super* :: *'a prog* ⇒ *cname* ⇒ *cname*
where *super P C* ≡ *fst* (*the* (*class P C*))

lemma *superI*:
(*C*,*D*) ∈ *subcls1 P* ⟹ *super P C* = *D*
by (*unfold super-def*) (*auto dest*: *subcls1D*)

primrec *the-Class* :: *ty* ⇒ *cname*
where
the-Class (*Class C*) = *C*

definition *sup* :: *'c prog* ⇒ *ty* ⇒ *ty* ⇒ *ty err*
where
sup P T$_1$ *T*$_2$ ≡
if is-refT T$_1$ ∧ *is-refT T*$_2$ *then*
OK (*if T*$_1$ = *NT then T*$_2$ *else*
if T$_2$ = *NT then T*$_1$ *else*
(*Class* (*exec-lub* (*subcls1 P*) (*super P*) (*the-Class T*$_1$) (*the-Class T*$_2$))))
else
(*if T*$_1$ = *T*$_2$ *then OK T*$_1$ *else Err*)

lemma *sup-def'*:
sup P = (λ*T*$_1$ *T*$_2$.
if is-refT T$_1$ ∧ *is-refT T*$_2$ *then*
OK (*if T*$_1$ = *NT then T*$_2$ *else*
if T$_2$ = *NT then T*$_1$ *else*
(*Class* (*exec-lub* (*subcls1 P*) (*super P*) (*the-Class T*$_1$) (*the-Class T*$_2$))))
else
(*if T*$_1$ = *T*$_2$ *then OK T*$_1$ *else Err*))
by (*simp add*: *sup-def fun-eq-iff*)

abbreviation
subtype :: *'c prog* ⇒ *ty* ⇒ *ty* ⇒ *bool*
where *subtype P* ≡ *widen P*

definition *esl* :: *'c prog* ⇒ *ty esl*
where
esl P ≡ (*types P*, *subtype P*, *sup P*)

lemma *is-class-is-subcls*:
wf-prog m P ⟹ *is-class P C* = *P* ⊢ *C* ≼* *Object*

lemma *subcls-antisym*:
⟦*wf-prog m P*; *P* ⊢ *C* ≼* *D*; *P* ⊢ *D* ≼* *C*⟧ ⟹ *C* = *D*

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

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

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

lemma *sup-subtype-greater*:
⟦ *wf-prog wf-mb P*; *is-type P t1*; *is-type P t2*; *sup P t1 t2* = *OK s* ⟧
⟹ *subtype P t1 s* ∧ *subtype P t2 s*
lemma *sup-subtype-smallest*:
⟦ *wf-prog wf-mb P*; *is-type P a*; *is-type P b*; *is-type P c*;
subtype P a c; *subtype P b c*; *sup P a b* = *OK d* ⟧
⟹ *subtype P d c*
lemma *sup-exists*:
⟦ *subtype P a c*; *subtype P b c* ⟧ ⟹ ∃ *T*. *sup P a b* = *OK T*
lemma *err-semilat-JType-esl*:
wf-prog wf-mb P ⟹ *err-semilat* (*esl P*)

end

4.15 The JVM Type System as Semilattice

theory *JVM-SemiType* **imports** *SemiType* **begin**

type-synonym ty_l = *ty err list*
type-synonym ty_s = *ty list*
type-synonym ty_i = $ty_s \times ty_l$
type-synonym ty_i' = ty_i *option*
type-synonym ty_m = ty_i' *list*
type-synonym ty_P = *mname* ⇒ *cname* ⇒ ty_m

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

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

definition *sl* :: *'c prog* ⇒ *nat* ⇒ *nat* ⇒ *ty$_i$' err sl*
where
sl P mxs mxl ≡
Err.sl(*Opt.esl*(*Product.esl* (*stk-esl P mxs*) (*Err.esl*(*loc-sl P mxl*))))

definition *states* :: *'c prog* ⇒ *nat* ⇒ *nat* ⇒ *ty$_i$' err set*
where *states P mxs mxl* ≡ *fst*(*sl P mxs mxl*)

definition *le* :: *'c prog* ⇒ *nat* ⇒ *nat* ⇒ *ty$_i$' err ord*
where
le P mxs mxl ≡ *fst*(*snd*(*sl P mxs mxl*))

definition *sup* :: *'c prog* ⇒ *nat* ⇒ *nat* ⇒ *ty$_i$' err binop*
where
sup P mxs mxl ≡ *snd*(*snd*(*sl P mxs mxl*))

definition *sup-ty-opt* :: [*'c prog*,*ty err*,*ty err*] ⇒ *bool*
(- ⊢ - ≤$_\top$ - [*71*,*71*,*71*] *70*)
where
sup-ty-opt P ≡ *Err.le* (*subtype P*)

definition *sup-state* :: [*'c prog*,*ty$_i$*,*ty$_i$*] ⇒ *bool*
(- ⊢ - ≤$_i$ - [*71*,*71*,*71*] *70*)
where
sup-state P ≡ *Product.le* (*Listn.le* (*subtype P*)) (*Listn.le* (*sup-ty-opt P*))

definition *sup-state-opt* :: [*'c prog*,*ty$_i$'*,*ty$_i$'*] ⇒ *bool*
(- ⊢ - ≤' - [*71*,*71*,*71*] *70*)
where
sup-state-opt P ≡ *Opt.le* (*sup-state P*)

abbreviation
sup-loc :: [*'c prog*,*ty$_l$*,*ty$_l$*] ⇒ *bool* (- ⊢ - [≤$_\top$] - [*71*,*71*,*71*] *70*)
where *P* ⊢ *LT* [≤$_\top$] *LT'* ≡ *list-all2* (*sup-ty-opt P*) *LT LT'*

notation (*ASCII*)
sup-ty-opt (- |- - <=*T* - [*71*,*71*,*71*] *70*) **and**
sup-state (- |- - <=*i* - [*71*,*71*,*71*] *70*) **and**
sup-state-opt (- |- - <=' - [*71*,*71*,*71*] *70*) **and**
sup-loc (- |- - [<=*T*] - [*71*,*71*,*71*] *70*)

4.15.1 Unfolding

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

lemma *JVM-le-conv*:
le P m n (*OK t1*) (*OK t2*) = *P* ⊢ *t1* ≤′ *t2*
lemma *JVM-le-Err-conv*:
le P m n = *Err.le* (*sup-state-opt P*)
lemma *err-le-unfold* [*iff*]:
Err.le r (*OK a*) (*OK b*) = *r a b*

4.15.2 Semilattice

lemma *order-sup-state-opt′* [*intro*, *simp*]:
wf-prog wf-mb P ⟹
order (*sup-state-opt P*) (*opt* ((⋃ {*nlists n* (*types P*) |*n*. *n* ≤ *mxs*}) × *nlists* (*Suc* (*length Ts* + mxl_0)) (*err* (*types P*))))

4.16 Effect of Instructions on the State Type

theory *Effect*
imports *JVM-SemiType ../JVM/JVMExceptions*
begin

— FIXME
locale *prog* =
fixes *P* :: ′*a prog*

locale *jvm-method* = *prog* +
fixes *mxs* :: *nat*
fixes mxl_0 :: *nat*
fixes *Ts* :: *ty list*
fixes T_r :: *ty*
fixes *is* :: *instr list*
fixes *xt* :: *ex-table*

fixes *mxl* :: *nat*
defines *mxl-def*: *mxl* ≡ *1*+*size Ts*+mxl_0

Program counter of successor instructions:

primrec *succs* :: *instr* ⇒ ty_i ⇒ *pc* ⇒ *pc list* **where**
succs (*Load idx*) *τ pc* = [*pc*+*1*]
| *succs* (*Store idx*) *τ pc* = [*pc*+*1*]
| *succs* (*Push v*) *τ pc* = [*pc*+*1*]
| *succs* (*Getfield F C*) *τ pc* = [*pc*+*1*]
| *succs* (*Putfield F C*) *τ pc* = [*pc*+*1*]
| *succs* (*New C*) *τ pc* = [*pc*+*1*]
| *succs* (*Checkcast C*) *τ pc* = [*pc*+*1*]
| *succs Pop τ pc* = [*pc*+*1*]
| *succs IAdd τ pc* = [*pc*+*1*]
| *succs CmpEq τ pc* = [*pc*+*1*]
| *succs-IfFalse*:
succs (*IfFalse b*) *τ pc* = [*pc*+*1*, *nat* (*int pc* + *b*)]
| *succs-Goto*:
succs (*Goto b*) *τ pc* = [*nat* (*int pc* + *b*)]
| *succs-Return*:
succs Return τ pc = []

| *succs-Invoke*:
  succs (*Invoke M n*) τ *pc* = (*if* (*fst* τ)!*n* = *NT then* [] *else* [*pc*+*1*])
| *succs-Throw*:
  succs Throw τ *pc* = []

Effect of instruction on the state type:

fun *the-class*:: *ty* ⇒ *cname* **where**
the-class (*Class C*) = *C*

fun *eff*$_i$ :: *instr* × '*m prog* × *ty*$_i$ ⇒ *ty*$_i$ **where**
eff$_i$-*Load*:
  eff$_i$ (*Load n*, *P*, (*ST*, *LT*)) = (*ok-val* (*LT* ! *n*) # *ST*, *LT*)
| *eff*$_i$-*Store*:
  eff$_i$ (*Store n*, *P*, (*T*#*ST*, *LT*)) = (*ST*, *LT*[*n*:= *OK T*])
| *eff*$_i$-*Push*:
  eff$_i$ (*Push v*, *P*, (*ST*, *LT*)) = (*the* (*typeof v*) # *ST*, *LT*)
| *eff*$_i$-*Getfield*:
  eff$_i$ (*Getfield F C*, *P*, (*T*#*ST*, *LT*)) = (*snd* (*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*

fun *is-relevant-class* :: *instr* ⇒ '*m prog* ⇒ *cname* ⇒ *bool* **where**
rel-Getfield:
  is-relevant-class (*Getfield F D*) = (λ*P C*. *P* ⊢ *NullPointer* ⪯* *C*)
| *rel-Putfield*:
  is-relevant-class (*Putfield F D*) = (λ*P C*. *P* ⊢ *NullPointer* ⪯* *C*)
| *rel-Checcast*:
  is-relevant-class (*Checkcast D*) = (λ*P C*. *P* ⊢ *ClassCast* ⪯* *C*)
| *rel-New*:
  is-relevant-class (*New D*) = (λ*P C*. *P* ⊢ *OutOfMemory* ⪯* *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*)

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

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

definition *xcpt-eff* :: *instr* ⇒ *'m prog* ⇒ *pc* ⇒ *ty*$_i$
⇒ *ex-table* ⇒ (*pc* × *ty*$_i$*'*) *list* **where**
xcpt-eff i P pc τ et = (*let* (*ST*,*LT*) = *τ in*
map (λ(*f*,*t*,*C*,*h*,*d*). (*h*, *Some* (*Class C*#*drop* (*size ST* − *d*) *ST*, *LT*))) (*relevant-entries P i pc et*))

definition *norm-eff* :: *instr* ⇒ *'m prog* ⇒ *nat* ⇒ *ty*$_i$ ⇒ (*pc* × *ty*$_i$*'*) *list* **where**
norm-eff i P pc τ = *map* (λ*pc'*. (*pc'*,*Some* (*eff*$_i$ (*i*,*P*,*τ*)))) (*succs i τ pc*)

definition *eff* :: *instr* ⇒ *'m prog* ⇒ *pc* ⇒ *ex-table* ⇒ *ty*$_i$*'* ⇒ (*pc* × *ty*$_i$*'*) *list* **where**
eff i P pc et t = (*case t of*
None ⇒ []
| *Some τ* ⇒ (*norm-eff i P pc τ*) @ (*xcpt-eff i P pc τ et*))

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

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

Conditions under which eff is applicable:

fun *app*$_i$ :: *instr* × *'m prog* × *pc* × *nat* × *ty* × *ty*$_i$ ⇒ *bool* **where**
app$_i$*-Load*:
app$_i$ (*Load n*, *P*, *pc*, *mxs*, *T*$_r$, (*ST*,*LT*)) =
(*n* < *length LT* ∧ *LT* ! *n* ≠ *Err* ∧ *length ST* < *mxs*)
| *app*$_i$*-Store*:
app$_i$ (*Store n*, *P*, *pc*, *mxs*, *T*$_r$, (*T*#*ST*, *LT*)) =
(*n* < *length LT*)
| *app*$_i$*-Push*:
app$_i$ (*Push v*, *P*, *pc*, *mxs*, *T*$_r$, (*ST*,*LT*)) =
(*length ST* < *mxs* ∧ *typeof v* ≠ *None*)
| *app*$_i$*-Getfield*:
app$_i$ (*Getfield F C*, *P*, *pc*, *mxs*, *T*$_r$, (*T*#*ST*, *LT*)) =
(∃ *T*$_f$. *P* ⊢ *C sees F*:*T*$_f$ *in C* ∧ *P* ⊢ *T* ≤ *Class C*)
| *app*$_i$*-Putfield*:
app$_i$ (*Putfield F C*, *P*, *pc*, *mxs*, *T*$_r$, (*T*$_1$#*T*$_2$#*ST*, *LT*)) =
(∃ *T*$_f$. *P* ⊢ *C sees F*:*T*$_f$ *in C* ∧ *P* ⊢ *T*$_2$ ≤ (*Class C*) ∧ *P* ⊢ *T*$_1$ ≤ *T*$_f$)
| *app*$_i$*-New*:
app$_i$ (*New C*, *P*, *pc*, *mxs*, *T*$_r$, (*ST*,*LT*)) =
(*is-class P C* ∧ *length ST* < *mxs*)
| *app*$_i$*-Checkcast*:
app$_i$ (*Checkcast C*, *P*, *pc*, *mxs*, *T*$_r$, (*T*#*ST*,*LT*)) =
(*is-class P C* ∧ *is-refT T*)
| *app*$_i$*-Pop*:
app$_i$ (*Pop*, *P*, *pc*, *mxs*, *T*$_r$, (*T*#*ST*,*LT*)) =

True
| app_i-*IAdd*:
app$_i$ (*IAdd*, *P*, *pc*, *mxs*, T_r, (T_1#T_2#*ST*,*LT*)) = (T_1 = T_2 ∧ T_1 = *Integer*)
| app_i-*CmpEq*:
app$_i$ (*CmpEq*, *P*, *pc*, *mxs*, T_r, (T_1#T_2#*ST*,*LT*)) =
(T_1 = T_2 ∨ *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*

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

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

lemma *app-Some*:
app i P mxs T_r *pc mpc xt* (*Some* τ) =
(*app*$_i$ (*i*,*P*,*pc*,*mxs*,T_r,τ) ∧ *xcpt-app i P pc mxs xt* τ ∧
(∀(*pc*',*s*') ∈ *set* (*eff i P pc xt* (*Some* τ)). *pc*' < *mpc*))
by (*simp add*: *app-def*)

locale *eff* = *jvm-method* +
fixes *eff*$_i$ **and** *app*$_i$ **and** *eff* **and** *app*
fixes *norm-eff* **and** *xcpt-app* **and** *xcpt-eff*

fixes *mpc*
defines *mpc* ≡ *size is*

defines *eff*$_i$ *i* τ ≡ *Effect*.*eff*$_i$ (*i*,*P*,τ)
notes *eff*$_i$-*simps* [*simp*] = *Effect*.*eff*$_i$.*simps* [**where** *P* = *P*, *folded eff*$_i$-*def*]

defines *app$_i$ i pc τ* ≡ *Effect.app$_i$* (*i*, *P*, *pc*, *mxs*, *T$_r$*, *τ*)
notes *app$_i$-simps* [*simp*] = *Effect.app$_i$.simps* [**where** *P*=*P* **and** *mxs*=*mxs* **and** *T$_r$*=*T$_r$*, *folded app$_i$-def*]

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

defines *norm-eff i pc τ* ≡ *Effect.norm-eff i P pc τ*
notes *norm-eff* = *Effect.norm-eff-def* [*of* - *P*, *folded norm-eff-def eff$_i$-def*]

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

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

defines *app i pc* ≡ *Effect.app i P mxs T$_r$ pc mpc xt*
notes *app* = *Effect.app-def* [*of* - *P mxs T$_r$* - *mpc xt*, *folded app-def xcpt-app-def app$_i$-def eff-def*]

lemma *length-cases2*:
assumes ⋀*LT*. *P* ([],*LT*)
assumes ⋀*l ST LT*. *P* (*l*#*ST*,*LT*)
shows *P s*
by (*cases s*, *cases fst s*) (*auto intro*!: *assms*)

lemma *length-cases3*:
assumes ⋀*LT*. *P* ([],*LT*)
assumes ⋀*l LT*. *P* ([*l*],*LT*)
assumes ⋀*l ST LT*. *P* (*l*#*ST*,*LT*)
shows *P s*
lemma *length-cases4*:
assumes ⋀*LT*. *P* ([],*LT*)
assumes ⋀*l LT*. *P* ([*l*],*LT*)
assumes ⋀*l l′ LT*. *P* ([*l*,*l′*],*LT*)
assumes ⋀*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*) = (∃ *ST LT*. *s* = (*ST*,*LT*) ∧ *idx* < *length LT* ∧ *LT*!*idx* ≠ *Err* ∧ *length ST* < *mxs*)
by (*cases s*, *simp*)

lemma *appStore*[*simp*]:
app$_i$ (*Store idx*,*P*,*pc*,*mxs*,*T$_r$*,*s*) = (∃ *ts ST LT*. *s* = (*ts*#*ST*,*LT*) ∧ *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:

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* ∧ *xcpt-app i P pc mxs xt' τ*)
by (*unfold xcpt-app-def*) *fastforce*

lemma *xcpt-eff-append* [*simp*]:
xcpt-eff i P pc τ (*xt*@*xt'*) = *xcpt-eff i P pc τ xt* @ *xcpt-eff i P pc τ xt'*
by (*unfold xcpt-eff-def*, *cases τ*) *simp*

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

end

4.17 Monotonicity of eff and app

theory *EffectMono* **imports** *Effect* **begin**

declare *not-Err-eq* [*iff*]

lemma *app*$_i$*-mono*:
assumes *wf*: *wf-prog p P*
assumes *less*: *P* ⊢ *τ* ≤$_i$ *τ'*
shows *app*$_i$ (*i*,*P*,*mxs*,*mpc*,*rT*,*τ'*) ⟹ *app*$_i$ (*i*,*P*,*mxs*,*mpc*,*rT*,*τ*)
lemma *succs-mono*:
assumes *wf*: *wf-prog p P* **and** *app*$_i$: *app*$_i$ (*i*,*P*,*mxs*,*mpc*,*rT*,*τ'*)
shows *P* ⊢ *τ* ≤$_i$ *τ'* ⟹ *set* (*succs i τ pc*) ⊆ *set* (*succs i τ' pc*)

lemma *app-mono*:
assumes *wf*: *wf-prog p P*
assumes *less'*: *P* ⊢ *τ* ≤' *τ'*
shows *app i P m rT pc mpc xt τ'* ⟹ *app i P m rT pc mpc xt τ*

lemma *eff*$_i$*-mono*:
assumes *wf*: *wf-prog p P*
assumes *less*: *P* ⊢ *τ* ≤$_i$ *τ'*
assumes *app*$_i$: *app i P m rT pc mpc xt* (*Some τ'*)
assumes *succs*: *succs i τ pc* ≠ [] *succs i τ' pc* ≠ []
shows *P* ⊢ *eff*$_i$ (*i*,*P*,*τ*) ≤$_i$ *eff*$_i$ (*i*,*P*,*τ'*)
end

4.18 The Bytecode Verifier

theory *BVSpec*
imports *Effect*
begin

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

definition
— The method type only contains declared classes:
check-types :: *'m prog* ⇒ *nat* ⇒ *nat* ⇒ *ty$_i$' err list* ⇒ *bool*
where
check-types P mxs mxl τs ≡ *set τs* ⊆ *states P mxs mxl*

— An instruction is welltyped if it is applicable and its effect
— is compatible with the type at all successor instructions:
definition
wt-instr :: [*'m prog*,*ty*,*nat*,*pc*,*ex-table*,*instr*,*pc*,*ty$_m$*] ⇒ *bool*
(-,-,-,-,- ⊢ -,- :: - [*60*,*0*,*0*,*0*,*0*,*0*,*0*,*61*] *60*)
where
P,*T*,*mxs*,*mpc*,*xt* ⊢ *i*,*pc* :: *τs* ≡
app i P mxs T pc mpc xt (*τs*!*pc*) ∧
(∀(*pc'*,*τ'*) ∈ *set* (*eff i P pc xt* (*τs*!*pc*)). *P* ⊢ *τ'* ≤' *τs*!*pc'*)

— The type at *pc=0* conforms to the method calling convention:
definition *wt-start* :: [*'m prog*,*cname*,*ty list*,*nat*,*ty$_m$*] ⇒ *bool*
where
wt-start P C Ts mxl$_0$ τs ≡
P ⊢ *Some* ([],*OK* (*Class C*)#*map OK Ts*@*replicate mxl$_0$ Err*) ≤' *τs*!*0*

— A method is welltyped if the body is not empty,
— if the method type covers all instructions and mentions
— declared classes only, if the method calling convention is respected, and
— if all instructions are welltyped.
definition *wt-method* :: [*'m prog*,*cname*,*ty list*,*ty*,*nat*,*nat*,*instr list*,
ex-table,*ty$_m$*] ⇒ *bool*
where
wt-method P C Ts T$_r$ mxs mxl$_0$ is xt τs ≡
0 < *size is* ∧ *size τs* = *size is* ∧
check-types P mxs (*1*+*size Ts*+*mxl$_0$*) (*map OK τs*) ∧
wt-start P C Ts mxl$_0$ τs ∧
(∀ *pc* < *size is*. *P*,*T$_r$*,*mxs*,*size is*,*xt* ⊢ *is*!*pc*,*pc* :: *τs*)

— A program is welltyped if it is wellformed and all methods are welltyped
definition *wf-jvm-prog-phi* :: *ty$_P$* ⇒ *jvm-prog* ⇒ *bool* (*wf'-jvm'-prog*$_-$)
where
wf-jvm-prog$_\Phi$ ≡
wf-prog (λ*P C* (*M*,*Ts*,*T$_r$*,(*mxs*,*mxl$_0$*,*is*,*xt*)).
wt-method P C Ts T$_r$ mxs mxl$_0$ is xt (Φ *C M*))

definition *wf-jvm-prog* :: *jvm-prog* ⇒ *bool*
where
wf-jvm-prog P ≡ ∃ Φ. *wf-jvm-prog*$_\Phi$ *P*

lemma *wt-jvm-progD*:
wf-jvm-prog$_\Phi$ *P* ⟹ ∃ *wt*. *wf-prog wt P*
lemma *wt-jvm-prog-impl-wt-instr*:
assumes *wf*: *wf-jvm-prog*$_\Phi$ *P* **and**
sees: *P* ⊢ *C sees M*:*Ts* → *T* = (*mxs*,*mxl$_0$*,*ins*,*xt*) *in C* **and**
pc: *pc* < *size ins*
shows *P*,*T*,*mxs*,*size ins*,*xt* ⊢ *ins*!*pc*,*pc* :: Φ *C M*
lemma *wt-jvm-prog-impl-wt-start*:

assumes *wf*: *wf-jvm-prog*$_\Phi$ *P* **and**
sees: *P* ⊢ *C sees M*:*Ts* → *T* = (*mxs*,*mxl*$_0$,*ins*,*xt*) *in C*
shows *0* < *size ins* ∧ *wt-start P C Ts mxl*$_0$ (Φ *C M*)

4.19 The Typing Framework for the JVM

theory *TF-JVM*
imports *../DFA/Typing-Framework-err EffectMono BVSpec*
begin

definition *exec* :: *jvm-prog* ⇒ *nat* ⇒ *ty* ⇒ *ex-table* ⇒ *instr list* ⇒ *ty*$_i$′ *err step-type*
where
exec G maxs rT et bs ≡
err-step (*size bs*) (λ*pc*. *app* (*bs*!*pc*) *G maxs rT pc* (*size bs*) *et*)
(λ*pc*. *eff* (*bs*!*pc*) *G pc et*)

locale *JVM-sl* =
fixes *P* :: *jvm-prog* **and** *mxs* **and** *mxl*$_0$ **and** *n*
fixes *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*

defines [*simp*]: *n* ≡ *size is*

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*)

4.19.1 Connecting JVM and Framework

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

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

lemma *singleton-nlists*:
(∃ *n*. [*Class* *C*] ∈ *nlists* *n* (*types* *P*) ∧ *n* ≤ *mxs*) = (*is-class* *P* *C* ∧ 0 < *mxs*)
by *auto*

lemma *set-drop-subset*:
set *xs* ⊆ *A* ⟹ *set* (*drop* *n* *xs*) ⊆ *A*
by (*auto* *dest*: *in-set-dropD*)

lemma *Suc-minus-minus-le*:
n < *mxs* ⟹ *Suc* (*n* − (*n* − *b*)) ≤ *mxs*
by *arith*

lemma *in-nlistsE*:
⟦ *xs* ∈ *nlists* *n* *A*; ⟦*size* *xs* = *n*; *set* *xs* ⊆ *A*⟧ ⟹ *P* ⟧ ⟹ *P*
by (*unfold* *nlists-def*) *blast*

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

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

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

lemma *conjI2*: ⟦ *A*; *A* ⟹ *B* ⟧ ⟹ *A* ∧ *B* **by** *blast*

lemma (**in** *JVM-sl*) *eff-mono*:
⟦*wf-prog* *p* *P*; *pc* < *length* *is*; *s* ⊑$_{sup\text{-}state\text{-}opt\ P}$ *t*; *app* *pc* *t*⟧
⟹ *set* (*eff* *pc* *s*) {⊑$_{sup\text{-}state\text{-}opt\ P}$} *set* (*eff* *pc* *t*)
lemma (**in** *JVM-sl*) *bounded-step*: *bounded* *step* (*size* *is*)
theorem (**in** *JVM-sl*) *step-mono*:
wf-prog *wf-mb* *P* ⟹ *mono* *r* *step* (*size* *is*) *A*

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

lemma (**in** *JVM-sl*) *wt-method-def2*:
wt-method *P* *C'* *Ts* T_r *mxs* mxl_0 *is* *xt* *τs* =
(*is* ≠ [] ∧

size τs = size is ∧
OK ' set τs ⊆ states P mxs mxl ∧
wt-start P C' Ts mxl₀ τs ∧
wt-app-eff (sup-state-opt P) app eff τs)

end

4.20 Typing and Dataflow Analysis Framework

theory *Typing-Framework-2* **imports** *Typing-Framework-1* **begin**

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

definition *is-bcv* :: *'s ord ⇒ 's ⇒ 's step-type ⇒ nat ⇒ 's set ⇒ ('s list ⇒ 's list) ⇒ bool*
where
is-bcv r T step n A bcv ⟷ (∀ τs₀ ∈ nlists n A.
(∀ p<n. (bcv τs₀)!p ≠ T) = (∃ τs ∈ nlists n A. τs₀ [⊑ᵣ] τs ∧ wt-step r T step τs))

end

4.21 Kildall for the JVM

theory *BVExec*
imports *../DFA/Abstract-BV TF-JVM ../DFA/Typing-Framework-2*
begin

definition *kiljvm* :: *jvm-prog ⇒ nat ⇒ nat ⇒ ty ⇒*
instr list ⇒ ex-table ⇒ tyᵢ' err list ⇒ tyᵢ' err list
where
kiljvm P mxs mxl Tᵣ is xt ≡
kildall (JVM-SemiType.le P mxs mxl) (JVM-SemiType.sup P mxs mxl)
(exec P mxs Tᵣ xt is)

definition *wt-kildall* :: *jvm-prog ⇒ cname ⇒ ty list ⇒ ty ⇒ nat ⇒ nat ⇒*
instr list ⇒ ex-table ⇒ bool
where
wt-kildall P C' Ts Tᵣ mxs mxl₀ is xt ≡
0 < size is ∧
(let first = Some ([],[OK (Class C')]@(map OK Ts)@(replicate mxl₀ Err));
start = OK first#(replicate (size is − 1) (OK None));
result = kiljvm P mxs (1+size Ts+mxl₀) Tᵣ is xt start
in ∀ n < size is. result!n ≠ Err)

definition *wf-jvm-progₖ* :: *jvm-prog ⇒ bool*
where
wf-jvm-progₖ P ≡
wf-prog (λP C' (M,Ts,Tᵣ,(mxs,mxl₀,is,xt)). wt-kildall P C' Ts Tᵣ mxs mxl₀ is xt) P

context *start-context*
begin

lemma *Cons-less-Conss3* [*simp*]:
x#xs [⊏ᵣ] y#ys = (x ⊏ᵣ y ∧ xs [⊑ᵣ] ys ∨ x = y ∧ xs [⊏ᵣ] ys)
apply (*unfold lesssub-def*)

apply *auto*
apply (*insert sup-state-opt-err*)
apply (*unfold lesssub-def lesub-def sup-state-opt-def sup-state-def sup-ty-opt-def*)
apply (*simp only*: *JVM-le-unfold*)
apply *fastforce*
done

lemma *acc-le-listI3* [*intro*!]:
acc r ⟹ *acc* (*Listn.le r*)
apply (*unfold acc-def*)
apply (*subgoal-tac*
wf(*UN n*. {(*ys*,*xs*). *size xs* = *n* ∧ *size ys* = *n* ∧ *xs* <-(*Listn.le r*) *ys*}))
apply (*erule wf-subset*)
apply (*blast intro*: *lesssub-lengthD*)
apply (*rule wf-UN*)
prefer *2*
apply (*rename-tac m n*)
apply (*case-tac m*=*n*)
apply *simp*
apply (*fast intro*!: *equals0I dest*: *not-sym*)
apply (*rename-tac n*)
apply (*induct-tac n*)
apply (*simp add*: *lesssub-def cong*: *conj-cong*)
apply (*rename-tac k*)
apply (*simp add*: *wf-eq-minimal del*: *r-def f-def step-def A-def*)
apply (*simp* (*no-asm*) *add*: *length-Suc-conv cong*: *conj-cong del*: *r-def f-def step-def A-def*)
apply *clarify*
apply (*rename-tac M m*)
apply (*case-tac* ∃ *x xs*. *size xs* = *k* ∧ *x*#*xs* ∈ *M*)
prefer *2*
apply (*erule thin-rl*)
apply (*erule thin-rl*)
apply *blast*
apply (*erule-tac x* = {*a*. ∃ *xs*. *size xs* = *k* ∧ *a*#*xs*:*M*} **in** *allE*)
apply (*erule impE*)
apply *blast*
apply (*thin-tac* ∃ *x xs*. *P x xs* **for** *P*)
apply *clarify*
apply (*rename-tac maxA xs*)
apply (*erule-tac x* = {*ys*. *size ys* = *size xs* ∧ *maxA*#*ys* ∈ *M*} **in** *allE*)
apply (*erule impE*)
apply *blast*
apply *clarify*
apply (*thin-tac m* ∈ *M*)
apply (*thin-tac maxA*#*xs* ∈ *M*)
apply (*rule bexI*)
prefer *2*
apply *assumption*
apply *clarify*
apply (*simp del*: *r-def f-def step-def A-def*)
apply *blast*
done

lemma *wf-jvm*: *wf* {(*ss'*, *ss*). *ss* [⊏$_r$] *ss'*}
apply (*insert acc-le-listI3 acc-JVM* [*OF wf*])
by (*simp add*: *acc-def r-def*)

lemma *iter-properties-bv*[*rule-format*]:
shows ⟦ ∀ *p*∈*w0*. *p* < *n*; *ss0* ∈ *nlists n A*; ∀ *p*<*n*. *p* ∉ *w0* ⟶ *stable r step ss0 p* ⟧ ⟹
iter f step ss0 w0 = (*ss'*,*w'*) ⟶
ss' ∈ *nlists n A* ∧ *stables r step ss'* ∧ *ss0* [⊑$_r$] *ss'* ∧
(∀ *ts*∈*nlists n A*. *ss0* [⊑$_r$] *ts* ∧ *stables r step ts* ⟶ *ss'* [⊑$_r$] *ts*)

lemma *kildall-properties-bv*:
shows ⟦ *ss0* ∈ *nlists n A* ⟧ ⟹
kildall r f step ss0 ∈ *nlists n A* ∧
stables r step (*kildall r f step ss0*) ∧
ss0 [⊑$_r$] *kildall r f step ss0* ∧
(∀ *ts*∈*nlists n A*. *ss0* [⊑$_r$] *ts* ∧ *stables r step ts* ⟶
kildall r f step ss0 [⊑$_r$] *ts*)

4.22 LBV for the JVM

theory *LBVJVM*
imports *../DFA/Abstract-BV TF-JVM*
begin

type-synonym *prog-cert* = *cname* ⇒ *mname* ⇒ *ty$_i$' err list*

definition *check-cert* :: *jvm-prog* ⇒ *nat* ⇒ *nat* ⇒ *nat* ⇒ *ty$_i$' err list* ⇒ *bool*
where
check-cert P mxs mxl n cert ≡ *check-types P mxs mxl cert* ∧ *size cert* = *n*+1 ∧
(∀ *i*<*n*. *cert*!*i* ≠ *Err*) ∧ *cert*!*n* = *OK None*

definition *lbvjvm* :: *jvm-prog* ⇒ *nat* ⇒ *nat* ⇒ *ty* ⇒ *ex-table* ⇒
ty$_i$' err list ⇒ *instr list* ⇒ *ty$_i$' err* ⇒ *ty$_i$' err*
where
lbvjvm P mxs maxr T$_r$ et cert bs ≡
wtl-inst-list bs cert (*JVM-SemiType.sup P mxs maxr*) (*JVM-SemiType.le P mxs maxr*) *Err* (*OK None*) (*exec P mxs T$_r$ et bs*) *0*

definition *wt-lbv* :: *jvm-prog* ⇒ *cname* ⇒ *ty list* ⇒ *ty* ⇒ *nat* ⇒ *nat* ⇒
ex-table ⇒ *ty$_i$' err list* ⇒ *instr list* ⇒ *bool*
where
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*)

definition *wt-jvm-prog-lbv* :: *jvm-prog* ⇒ *prog-cert* ⇒ *bool*
where
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*

definition *mk-cert* :: *jvm-prog* ⇒ *nat* ⇒ *ty* ⇒ *ex-table* ⇒ *instr list*
⇒ ty_m ⇒ ty_i' *err list*
where
mk-cert P mxs T_r *et bs phi* ≡ *make-cert* (*exec P mxs* T_r *et bs*) (*map OK phi*) (*OK None*)

definition *prg-cert* :: *jvm-prog* ⇒ ty_P ⇒ *prog-cert*
where
prg-cert P phi C mn ≡ *let* (*C*,*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* ∈ *nlists* (*size is*) *A*. *wt-step r Err step* τ*s* ∧ *OK first* ⊑$_r$ 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 ⟹ *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

4.23 BV Type Safety Invariant

theory *BVConform*
imports *BVSpec* ../*JVM*/*JVMExec* ../*Common*/*Conform*
begin

definition *confT* :: *'c prog* ⇒ *heap* ⇒ *val* ⇒ *ty err* ⇒ *bool*
(-,- ⊢ - :≤$_\top$ - [*51*,*51*,*51*,*51*] *50*)
where
P,*h* ⊢ *v* :≤$_\top$ *E* ≡ *case E of Err* ⇒ *True* | *OK T* ⇒ *P*,*h* ⊢ *v* :≤ *T*

notation (*ASCII*)
confT (-,- |- - :<=*T* - [*51*,*51*,*51*,*51*] *50*)

abbreviation
confTs :: *'c prog* ⇒ *heap* ⇒ *val list* ⇒ ty_l ⇒ *bool*

(-,- ⊢ - [:≤⊤] - [51,51,51,51] 50) **where**
P,h ⊢ vs [:≤⊤] Ts ≡ list-all2 (confT P h) vs Ts

notation (*ASCII*)
confTs (-,- |- - [:<=T] - [51,51,51,51] 50)

definition *conf-f* :: *jvm-prog ⇒ heap ⇒ ty_i ⇒ bytecode ⇒ frame ⇒ bool*
where
conf-f P h ≡ λ(ST,LT) is (stk,loc,C,M,pc).
P,h ⊢ stk [:≤] ST ∧ P,h ⊢ loc [:≤⊤] LT ∧ pc < size is

lemma *conf-f-def2*:
conf-f P h (ST,LT) is (stk,loc,C,M,pc) ≡
P,h ⊢ stk [:≤] ST ∧ P,h ⊢ loc [:≤⊤] LT ∧ pc < size is
by (*simp add*: *conf-f-def*)

primrec *conf-fs* :: [*jvm-prog,heap,ty_P,mname,nat,ty,frame list*] ⇒ *bool*
where
conf-fs P h Φ M₀ n₀ T₀ [] = True
| *conf-fs P h Φ M₀ n₀ T₀ (f#frs) =*
(let (stk,loc,C,M,pc) = f in
(∃ ST LT Ts T mxs mxl₀ is xt.
Φ C M ! pc = Some (ST,LT) ∧
(P ⊢ C sees M:Ts → T = (mxs,mxl₀,is,xt) in C) ∧
(∃ D Ts' T' m D'.
is!pc = (Invoke M₀ n₀) ∧ ST!n₀ = Class D ∧
P ⊢ D sees M₀:Ts' → T' = m in D' ∧ P ⊢ T₀ ≤ T') ∧
conf-f P h (ST, LT) is f ∧ conf-fs P h Φ M (size Ts) T frs))

definition *correct-state* :: [*jvm-prog,ty_P,jvm-state*] ⇒ *bool* (-,- ⊢ - √ [61,0,0] 61)
where
correct-state P Φ ≡ λ(xp,h,frs).
case xp of
None ⇒ (case frs of
[] ⇒ True
| (f#fs) ⇒ P⊢ h√ ∧
(let (stk,loc,C,M,pc) = f
in ∃ Ts T mxs mxl₀ is xt τ.
(P ⊢ C sees M:Ts→T = (mxs,mxl₀,is,xt) in C) ∧
Φ C M ! pc = Some τ ∧
conf-f P h τ is f ∧ conf-fs P h Φ M (size Ts) T fs))
| *Some x ⇒ frs = []*

notation
correct-state (-,- |- - [ok] [61,0,0] 61)

4.23.1 Values and ⊤

lemma *confT-Err* [*iff*]: *P,h ⊢ x :≤⊤ Err*
by (*simp add*: *confT-def*)

lemma *confT-OK* [*iff*]: *P,h ⊢ x :≤⊤ OK T = (P,h ⊢ x :≤ 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*)

4.23.2 Stack and Registers

lemmas *confTs-Cons1* [*iff*] = *list-all2-Cons1* [*of confT P h*] **for** *P h*

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

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

4.23.3 correct-frames

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

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

4.24 BV Type Safety Proof

theory *BVSpecTypeSafe*
imports *BVConform*

begin

This theory contains proof that the specification of the bytecode verifier only admits type safe programs.

4.24.1 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*

4.24.2 Exception Handling

For the *Invoke* instruction the BV has checked all handlers that guard the current *pc*.

lemma *Invoke-handlers*:
match-ex-table P C pc xt = Some (pc′,d′) ⟹
∃(f,t,D,h,d) ∈ set (relevant-entries P (Invoke n M) pc xt).
P ⊢ C ⪯ D ∧ pc ∈ {f..<t} ∧ pc′ = h ∧ d′ = d*
by (*induct xt*) (*auto simp add*: *relevant-entries-def matches-ex-entry-def*
is-relevant-entry-def split: *if-split-asm*)

We can prove separately that the recursive search for exception handlers (*find-handler*) in the frame stack results in a conforming state (if there was no matching exception handler in the current frame). We require that the exception is a valid heap address, and that the state before the exception occurred conforms.

term *find-handler*
lemma *uncaught-xcpt-correct*:
assumes *wt*: *wf-jvm-prog$_\Phi$ P*
assumes *h*: *h xcp = Some obj*
shows *⋀f. P,Φ ⊢ (None, h, f#frs)√ ⟹ P,Φ ⊢ (find-handler P xcp h frs) √*
(**is** *⋀f. ?correct (None, h, f#frs) ⟹ ?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-instr (ins!pc) P h stk vars Cl M pc frs) = Some xcp;
P,T,mxs,size ins,xt ⊢ ins!pc,pc :: Φ C M;
P,Φ ⊢ (None, h, (stk,loc,C,M,pc)#frs)√ ⟧
⟹ ∃obj. h xcp = Some obj
(**is** *⟦ ?xcpt; ?wt; ?correct ⟧ ⟹ ?thesis*)
lemma *conf-sys-xcpt*:
⟦preallocated h; C ∈ sys-xcpts⟧ ⟹ P,h ⊢ Addr (addr-of-sys-xcpt C) :≤ Class C
by (*auto elim*: *preallocatedE*)

lemma *match-ex-table-SomeD*:
match-ex-table P C pc xt = Some (pc′,d′) ⟹
∃(f,t,D,h,d) ∈ set xt. matches-ex-entry P C pc (f,t,D,h,d) ∧ h = pc′ ∧ d=d′
by (*induct xt*) (*auto split*: *if-split-asm*)

Finally we can state that, whenever an exception occurs, the next state always conforms:

lemma *xcpt-correct*:

fixes σ' :: *jvm-state*
assumes *wtp*: *wf-jvm-prog*$_\Phi$ *P*
assumes *meth*: *P* ⊢ *C sees M*:*Ts*→*T*=(*mxs*,*mxl*$_0$,*ins*,*xt*) *in C*
assumes *wt*: *P*,*T*,*mxs*,*size ins*,*xt* ⊢ *ins*!*pc*,*pc* :: Φ *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*,Φ ⊢ (*None*, *h*, (*stk*,*loc*,*C*,*M*,*pc*)#*frs*)√
shows *P*,Φ ⊢ σ'√

4.24.3 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*:
fixes σ' :: *jvm-state*
assumes *wtprog*: *wf-jvm-prog*$_\Phi$ *P*
assumes *meth-C*: *P* ⊢ *C sees M*:*Ts*→*T*=(*mxs*,*mxl*$_0$,*ins*,*xt*) *in C*
assumes *ins*: *ins* ! *pc* = *Invoke M' n*
assumes *wti*: *P*,*T*,*mxs*,*size ins*,*xt* ⊢ *ins*!*pc*,*pc* :: Φ *C M*
assumes *σ'*: *Some* σ' = *exec* (*P*, *None*, *h*, (*stk*,*loc*,*C*,*M*,*pc*)#*frs*)
assumes *approx*: *P*,Φ ⊢ (*None*, *h*, (*stk*,*loc*,*C*,*M*,*pc*)#*frs*)√
assumes *no-xcp*: *fst* (*exec-instr* (*ins*!*pc*) *P h stk loc C M pc frs*) = *None*
shows *P*,Φ ⊢ σ'√
declare *list-all2-Cons2* [*iff*]

lemma *Return-correct*:
fixes σ' :: *jvm-state*
assumes *wt-prog*: *wf-jvm-prog*$_\Phi$ *P*
assumes *meth*: *P* ⊢ *C sees M*:*Ts*→*T*=(*mxs*,*mxl*$_0$,*ins*,*xt*) *in C*
assumes *ins*: *ins* ! *pc* = *Return*
assumes *wt*: *P*,*T*,*mxs*,*size ins*,*xt* ⊢ *ins*!*pc*,*pc* :: Φ *C M*
assumes *s'*: *Some* σ' = *exec* (*P*, *None*, *h*, (*stk*,*loc*,*C*,*M*,*pc*)#*frs*)
assumes *correct*: *P*,Φ ⊢ (*None*, *h*, (*stk*,*loc*,*C*,*M*,*pc*)#*frs*)√

shows *P*,Φ ⊢ σ'√
declare *sup-state-opt-any-Some* [*iff*]
declare *not-Err-eq* [*iff*]

lemma *Load-correct*:
⟦ *wf-prog wt P*;
P ⊢ *C sees M*:*Ts*→*T*=(*mxs*,*mxl*$_0$,*ins*,*xt*) *in C*;
ins!*pc* = *Load idx*;
P,*T*,*mxs*,*size ins*,*xt* ⊢ *ins*!*pc*,*pc* :: Φ *C M*;
Some σ' = *exec* (*P*, *None*, *h*, (*stk*,*loc*,*C*,*M*,*pc*)#*frs*);
P,Φ ⊢ (*None*, *h*, (*stk*,*loc*,*C*,*M*,*pc*)#*frs*)√ ⟧
⟹ *P*,Φ ⊢ σ'√
by (*fastforce dest*: *sees-method-fun* [*of* - *C*] *elim*!: *confTs-confT-sup*)

declare [[*simproc del*: *list-to-set-comprehension*]]

lemma *Store-correct*:
⟦ *wf-prog wt P*;
P ⊢ *C sees M*:*Ts*→*T*=(*mxs*,*mxl*$_0$,*ins*,*xt*) *in C*;
ins!*pc* = *Store idx*;
P,*T*,*mxs*,*size ins*,*xt* ⊢ *ins*!*pc*,*pc* :: Φ *C M*;
Some σ′ = *exec* (*P*, *None*, *h*, (*stk*,*loc*,*C*,*M*,*pc*)#*frs*);
P,Φ ⊢ (*None*, *h*, (*stk*,*loc*,*C*,*M*,*pc*)#*frs*)√ ⟧
⟹ *P*,Φ ⊢ *σ′*√

lemma *Push-correct*:
⟦ *wf-prog wt P*;
P ⊢ *C sees M*:*Ts*→*T*=(*mxs*,*mxl*$_0$,*ins*,*xt*) *in C*;
ins!*pc* = *Push v*;
P,*T*,*mxs*,*size ins*,*xt* ⊢ *ins*!*pc*,*pc* :: Φ *C M*;
Some σ′ = *exec* (*P*, *None*, *h*, (*stk*,*loc*,*C*,*M*,*pc*)#*frs*);
P,Φ ⊢ (*None*, *h*, (*stk*,*loc*,*C*,*M*,*pc*)#*frs*)√ ⟧
⟹ *P*,Φ ⊢ *σ′*√

lemma *Cast-conf2*:
⟦ *wf-prog ok P*; *P*,*h* ⊢ *v* :≤ *T*; *is-refT T*; *cast-ok P C h v*;
P ⊢ *Class C* ≤ *T′*; *is-class P C*⟧
⟹ *P*,*h* ⊢ *v* :≤ *T′*

lemma *Checkcast-correct*:
⟦ *wf-jvm-prog*$_\Phi$ *P*;
P ⊢ *C sees M*:*Ts*→*T*=(*mxs*,*mxl*$_0$,*ins*,*xt*) *in C*;
ins!*pc* = *Checkcast D*;
P,*T*,*mxs*,*size ins*,*xt* ⊢ *ins*!*pc*,*pc* :: Φ *C M*;
Some σ′ = *exec* (*P*, *None*, *h*, (*stk*,*loc*,*C*,*M*,*pc*)#*frs*) ;
P,Φ ⊢ (*None*, *h*, (*stk*,*loc*,*C*,*M*,*pc*)#*frs*)√;
fst (*exec-instr* (*ins*!*pc*) *P h stk loc C M pc frs*) = *None* ⟧
⟹ *P*,Φ ⊢ *σ′*√
declare *split-paired-All* [*simp del*]

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

lemma *Getfield-correct*:
fixes *σ′* :: *jvm-state*
assumes *wf*: *wf-prog wt P*
assumes *mC*: *P* ⊢ *C sees M*:*Ts*→*T*=(*mxs*,*mxl*$_0$,*ins*,*xt*) *in C*
assumes *i*: *ins*!*pc* = *Getfield F D*
assumes *wt*: *P*,*T*,*mxs*,*size ins*,*xt* ⊢ *ins*!*pc*,*pc* :: Φ *C M*
assumes *s′*: *Some σ′* = *exec* (*P*, *None*, *h*, (*stk*,*loc*,*C*,*M*,*pc*)#*frs*)
assumes *cf*: *P*,Φ ⊢ (*None*, *h*, (*stk*,*loc*,*C*,*M*,*pc*)#*frs*)√
assumes *xc*: *fst* (*exec-instr* (*ins*!*pc*) *P h stk loc C M pc frs*) = *None*

shows *P*,Φ ⊢ *σ′*√
lemma *Putfield-correct*:
fixes *σ′* :: *jvm-state*
assumes *wf*: *wf-prog wt P*
assumes *mC*: *P* ⊢ *C sees M*:*Ts*→*T*=(*mxs*,*mxl*$_0$,*ins*,*xt*) *in C*
assumes *i*: *ins*!*pc* = *Putfield F D*
assumes *wt*: *P*,*T*,*mxs*,*size ins*,*xt* ⊢ *ins*!*pc*,*pc* :: Φ *C M*
assumes *s′*: *Some σ′* = *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-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*:
fixes σ' :: *jvm-state*
assumes *wf*: *wf-prog wt P*
assumes *meth*: $P \vdash C$ *sees* M:$Ts\to T$=(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:$Ts\to T$=(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:$Ts\to T$=(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:$Ts\to T$=(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:$Ts\to T$=(mxs,mxl_0,ins,xt) *in C*;
ins ! *pc* = *Pop*;
$P,T,mxs,size\ ins,xt \vdash ins!pc,pc :: \Phi\ C\ M$;

Some $\sigma' = exec\ (P,\ None,\ h,\ (stk,loc,C,M,pc)\#frs)$;
$P,\Phi \vdash (None,\ h,\ (stk,loc,C,M,pc)\#frs)\surd$ ⟧
⟹ $P,\Phi \vdash \sigma'\surd$

lemma *IAdd-correct*:
⟦ *wf-prog wt P*;
$P \vdash C$ *sees* $M:Ts\to T=(mxs,mxl_0,ins,xt)$ *in* C;
$ins\ !\ pc = IAdd$;
$P,T,mxs,size\ ins,xt \vdash ins!pc,pc :: \Phi\ C\ M$;
Some $\sigma' = exec\ (P,\ None,\ h,\ (stk,loc,C,M,pc)\#frs)$;
$P,\Phi \vdash (None,\ h,\ (stk,loc,C,M,pc)\#frs)\surd$ ⟧
⟹ $P,\Phi \vdash \sigma'\surd$

lemma *Throw-correct*:
⟦ *wf-prog wt P*;
$P \vdash C$ *sees* $M:Ts\to T=(mxs,mxl_0,ins,xt)$ *in* C;
$ins\ !\ pc = Throw$;
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 $(exec\text{-}instr\ (ins!pc)\ P\ h\ stk\ loc\ C\ M\ pc\ frs) = None$ ⟧
⟹ $P,\Phi \vdash \sigma'\surd$
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*:
⟦ $wf\text{-}jvm\text{-}prog_\Phi\ P$;
$P \vdash C$ *sees* $M:Ts\to T=(mxs,mxl_0,ins,xt)$ *in* C;
Some $\sigma' = exec\ (P,\ None,\ h,\ (stk,loc,C,M,pc)\#frs)$;
$P,\Phi \vdash (None,\ h,\ (stk,loc,C,M,pc)\#frs)\surd$ ⟧
⟹ $P,\Phi \vdash \sigma'\surd$

4.24.4 Main

lemma *correct-state-impl-Some-method*:
$P,\Phi \vdash (None,\ h,\ (stk,loc,C,M,pc)\#frs)\surd$
⟹ $\exists m\ Ts\ T.\ P \vdash C$ *sees* $M:Ts\to T = m$ *in* C
by *fastforce*

lemma *BV-correct-1* [*rule-format*]:
$\bigwedge\sigma.$ ⟦ $wf\text{-}jvm\text{-}prog_\Phi\ P$; $P,\Phi \vdash \sigma\surd$⟧ ⟹ $P \vdash \sigma\ -jvm\to_1\ \sigma' \longrightarrow P,\Phi \vdash \sigma'\surd$

theorem *progress*:
⟦ $xp=None$; $frs\neq[]$ ⟧ ⟹ $\exists\sigma'.\ P \vdash (xp,h,frs)\ -jvm\to_1\ \sigma'$
by (*clarsimp simp add*: *exec-1-iff neq-Nil-conv split-beta*
simp del: *split-paired-Ex*)

lemma *progress-conform*:
⟦$wf\text{-}jvm\text{-}prog_\Phi\ P$; $P,\Phi \vdash (xp,h,frs)\surd$; $xp=None$; $frs\neq[]$⟧
⟹ $\exists\sigma'.\ P \vdash (xp,h,frs)\ -jvm\to_1\ \sigma' \wedge P,\Phi \vdash \sigma'\surd$
theorem *BV-correct* [*rule-format*]:
⟦ $wf\text{-}jvm\text{-}prog_\Phi\ P$; $P \vdash \sigma\ -jvm\to\ \sigma'$ ⟧ ⟹ $P,\Phi \vdash \sigma\surd \longrightarrow P,\Phi \vdash \sigma'\surd$

lemma *hconf-start*:
assumes *wf*: *wf-prog wf-mb P*
shows $P \vdash (\textit{start-heap}\ P)\ \surd$
lemma *BV-correct-initial*:
shows $[\![\ \textit{wf-jvm-prog}_\Phi\ P;\ P \vdash C\ \textit{sees}\ M{:}[]\to T = m\ \textit{in}\ C\ ]\!]$
$\Longrightarrow P,\Phi \vdash \textit{start-state}\ P\ C\ M\ \surd$
theorem *typesafe*:
assumes *welltyped*: $\textit{wf-jvm-prog}_\Phi\ P$
assumes *main-method*: $P \vdash C\ \textit{sees}\ M{:}[]\to T = m\ \textit{in}\ C$
shows $P \vdash \textit{start-state}\ P\ C\ M\ -\textit{jvm}\to \sigma \implies P,\Phi \vdash \sigma\ \surd$
end

4.25 Welltyped Programs produce no Type Errors

theory *BVNoTypeError*
imports *../JVM/JVMDefensive BVSpecTypeSafe*
begin

lemma *has-methodI*:
$P \vdash C\ \textit{sees}\ M{:}Ts\to T = m\ \textit{in}\ D \Longrightarrow P \vdash C\ \textit{has}\ M$
by (*unfold has-method-def*) *blast*

Some simple lemmas about the type testing functions of the defensive JVM:

lemma *typeof-NoneD* [*simp*,*dest*]: $\textit{typeof}\ v = \textit{Some}\ x \Longrightarrow \neg \textit{is-Addr}\ v$
by (*cases v*) *auto*

lemma *is-Ref-def2*:
$\textit{is-Ref}\ v = (v = \textit{Null} \vee (\exists\, a.\ v = \textit{Addr}\ a))$
by (*cases v*) (*auto simp add*: *is-Ref-def*)

lemma [*iff*]: *is-Ref Null* **by** (*simp add*: *is-Ref-def2*)

lemma *is-RefI* [*intro*, *simp*]: $P,h \vdash v :\leq T \Longrightarrow \textit{is-refT}\ T \Longrightarrow \textit{is-Ref}\ v$
lemma *is-IntgI* [*intro*, *simp*]: $P,h \vdash v :\leq \textit{Integer} \Longrightarrow \textit{is-Intg}\ v$
lemma *is-BoolI* [*intro*, *simp*]: $P,h \vdash v :\leq \textit{Boolean} \Longrightarrow \textit{is-Bool}\ v$
declare *defs1* [*simp del*]

lemma *wt-jvm-prog-states*:
$[\![\ \textit{wf-jvm-prog}_\Phi\ P;\ P \vdash C\ \textit{sees}\ M{:}\ Ts\to T = (\textit{mxs},\ \textit{mxl},\ \textit{ins},\ \textit{et})\ \textit{in}\ C;$
$\Phi\ C\ M\ !\ pc = \tau;\ pc < \textit{size ins}\ ]\!]$
$\Longrightarrow \textit{OK}\ \tau \in \textit{states}\ P\ \textit{mxs}\ (1+\textit{size}\ Ts+\textit{mxl})$

The main theorem: welltyped programs do not produce type errors if they are started in a conformant state.

theorem *no-type-error*:
fixes σ :: *jvm-state*
assumes *welltyped*: $\textit{wf-jvm-prog}_\Phi\ P$ **and** *conforms*: $P,\Phi \vdash \sigma\ \surd$
shows $\textit{exec-d}\ P\ \sigma \neq \textit{TypeError}$

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*:
$\textit{wf-jvm-prog}_\Phi\ P \Longrightarrow P,\Phi \vdash \sigma\ \surd \Longrightarrow P \vdash \sigma\ -\textit{jvm}\to \sigma'$

$\Longrightarrow P \vdash (Normal\ \sigma)\ -jvmd\rightarrow\ (Normal\ \sigma')$

As corollary we get that the aggressive and the defensive machine are equivalent for well-typed programs (if started in a conformant state or in the canonical start state)

corollary *welltyped-commutes*:
fixes σ :: *jvm-state*
assumes *wf*: *wf-jvm-prog*$_\Phi$ P **and** *conforms*: $P,\Phi \vdash \sigma\ \surd$
shows $P \vdash (Normal\ \sigma)\ -jvmd\rightarrow\ (Normal\ \sigma') = P \vdash \sigma\ -jvm\rightarrow\ \sigma'$
apply *rule*
apply (*erule defensive-imp-aggressive*)
apply (*erule welltyped-aggressive-imp-defensive* [*OF wf conforms*])
done

corollary *welltyped-initial-commutes*:
assumes *wf*: *wf-jvm-prog P*
assumes *meth*: $P \vdash C\ sees\ M{:}[]\rightarrow T = b\ in\ C$
defines *start*: $\sigma \equiv start\text{-}state\ P\ C\ M$
shows $P \vdash (Normal\ \sigma)\ -jvmd\rightarrow\ (Normal\ \sigma') = P \vdash \sigma\ -jvm\rightarrow\ \sigma'$
proof −
from *wf* **obtain** Φ **where** *wf'*: *wf-jvm-prog*$_\Phi$ P **by** (*auto simp*: *wf-jvm-prog-def*)
from *this meth* **have** $P,\Phi \vdash \sigma\ \surd$ **unfolding** *start* **by** (*rule BV-correct-initial*)
with *wf'* **show** *?thesis* **by** (*rule welltyped-commutes*)
qed

lemma *not-TypeError-eq* [*iff*]:
$x \neq TypeError = (\exists\, t.\ x = Normal\ t)$
by (*cases x*) *auto*

locale *cnf* =
fixes P **and** Φ **and** σ
assumes *wf*: *wf-jvm-prog*$_\Phi$ P
assumes *cnf*: *correct-state P* Φ σ

theorem (**in** *cnf*) *no-type-errors*:
$P \vdash (Normal\ \sigma)\ -jvmd\rightarrow\ \sigma' \Longrightarrow \sigma' \neq TypeError$
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-eq*)
done

locale *start* =
fixes P **and** C **and** M **and** σ **and** T **and** b
assumes *wf*: *wf-jvm-prog P*
assumes *sees*: $P \vdash C\ sees\ M{:}[]\rightarrow T = b\ in\ C$
defines $\sigma \equiv Normal\ (start\text{-}state\ P\ C\ M)$

corollary (**in** *start*) *bv-no-type-error*:
shows $P \vdash \sigma\ -jvmd\rightarrow\ \sigma' \Longrightarrow \sigma' \neq TypeError$
proof −
from *wf* **obtain** Φ **where** $wf\text{-}jvm\text{-}prog_\Phi\ P$ **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 $P \vdash \sigma\ -jvmd\rightarrow\ \sigma'$
ultimately show *?thesis* **by** (*unfold* σ*-def*) (*rule cnf.no-type-errors*)
qed

end

4.26 Example Welltypings

theory *BVExample*
imports *../JVM/JVMListExample BVSpecTypeSafe BVExec*
HOL−Library.Code-Target-Numeral
begin

This theory shows type correctness of the example program in section 3.7 (p. 72) by explicitly providing a welltyping. It also shows that the start state of the program conforms to the welltyping; hence type safe execution is guaranteed.

4.26.1 Setup

lemma *distinct-classes'*:
list-name ≠ *test-name*
list-name ≠ *Object*
list-name ≠ *ClassCast*
list-name ≠ *OutOfMemory*
list-name ≠ *NullPointer*
test-name ≠ *Object*
test-name ≠ *OutOfMemory*
test-name ≠ *ClassCast*
test-name ≠ *NullPointer*
ClassCast ≠ *NullPointer*
ClassCast ≠ *Object*
NullPointer ≠ *Object*
OutOfMemory ≠ *ClassCast*
OutOfMemory ≠ *NullPointer*
OutOfMemory ≠ *Object*
by (*simp-all add*: *list-name-def test-name-def Object-def NullPointer-def*
OutOfMemory-def ClassCast-def)

lemmas *distinct-classes* = *distinct-classes' distinct-classes'* [*symmetric*]

lemma *distinct-fields*:
val-name ≠ *next-name*
next-name ≠ *val-name*
by (*simp-all add*: *val-name-def next-name-def*)

Abbreviations for definitions we will have to use often in the proofs below:

lemmas *system-defs = SystemClasses-def ObjectC-def NullPointerC-def OutOfMemoryC-def ClassCastC-def*
lemmas *class-defs = list-class-def test-class-def*

These auxiliary proofs are for efficiency: class lookup, subclass relation, method and field lookup are computed only once:

lemma *class-Object* [*simp*]:
class E Object = Some (*undefined*, [],[])
by (*simp add*: *class-def system-defs E-def*)

lemma *class-NullPointer* [*simp*]:
class E NullPointer = Some (*Object*, [], [])
by (*simp add*: *class-def system-defs E-def distinct-classes*)

lemma *class-OutOfMemory* [*simp*]:
class E OutOfMemory = Some (*Object*, [], [])
by (*simp add*: *class-def system-defs E-def distinct-classes*)

lemma *class-ClassCast* [*simp*]:
class E ClassCast = Some (*Object*, [], [])
by (*simp add*: *class-def system-defs E-def distinct-classes*)

lemma *class-list* [*simp*]:
class E list-name = Some list-class
by (*simp add*: *class-def system-defs E-def distinct-classes*)

lemma *class-test* [*simp*]:
class E test-name = Some test-class
by (*simp add*: *class-def system-defs E-def distinct-classes*)

lemma *E-classes* [*simp*]:
{*C*. *is-class E C*} = {*list-name*, *test-name*, *NullPointer*, *ClassCast*, *OutOfMemory*, *Object*}
by (*auto simp add*: *is-class-def class-def system-defs E-def class-defs*)

The subclass releation spelled out:

lemma *subcls1*:
subcls1 E = {(*list-name*,*Object*), (*test-name*,*Object*), (*NullPointer*, *Object*), (*ClassCast*, *Object*), (*OutOfMemory*, *Object*)}

The subclass relation is acyclic; hence its converse is well founded:

lemma *notin-rtrancl*:
(*a*,*b*) ∈ r^* ⟹ *a* ≠ *b* ⟹ (⋀*y*. (*a*,*y*) ∉ *r*) ⟹ *False*
by (*auto elim*: *converse-rtranclE*)

lemma *acyclic-subcls1-E*: *acyclic* (*subcls1 E*)
lemma *wf-subcls1-E*: *wf* ((*subcls1 E*)$^{-1}$)

Method and field lookup:

lemma *method-append* [*simp*]:
method E list-name append-name =
(*list-name*, [*Class list-name*], *Void*, *3*, *0*, *append-ins*, [(*1*, *2*, *NullPointer*, *7*, *0*)])

lemma *method-makelist* [*simp*]:
method E test-name makelist-name =
(*test-name*, [], *Void*, *3*, *2*, *make-list-ins*, [])
lemma *field-val* [*simp*]:
field E list-name val-name = (*list-name*, *Integer*)
lemma *field-next* [*simp*]:
field E list-name next-name = (*list-name*, *Class list-name*)
lemma [*simp*]: *fields E Object* = []
by (*fastforce intro*: *fields-def2 Fields.intros*)

lemma [*simp*]: *fields E NullPointer* = []
by (*fastforce simp add*: *distinct-classes intro*: *fields-def2 Fields.intros*)

lemma [*simp*]: *fields E ClassCast* = []
by (*fastforce simp add*: *distinct-classes intro*: *fields-def2 Fields.intros*)

lemma [*simp*]: *fields E OutOfMemory* = []
by (*fastforce simp add*: *distinct-classes intro*: *fields-def2 Fields.intros*)

lemma [*simp*]: *fields E test-name* = []
lemmas [*simp*] = *is-class-def*

4.26.2 Program structure

The program is structurally wellformed:

lemma *wf-struct*:
wf-prog (λ*G C mb*. *True*) *E* (**is** *wf-prog ?mb E*)

4.26.3 Welltypings

We show welltypings of the methods *append-name* in class *list-name*, and *makelist-name* in class *test-name*:

lemmas *eff-simps* [*simp*] = *eff-def norm-eff-def xcpt-eff-def*

definition *phi-append* :: ty_m (φ_a)
where
$\varphi_a \equiv$ *map* (λ(*x*,*y*). *Some* (*x*, *map OK y*)) [
([], [*Class list-name*, *Class list-name*]),
([*Class list-name*], [*Class list-name*, *Class list-name*]),
([*Class list-name*], [*Class list-name*, *Class list-name*]),
([*Class list-name*, *Class list-name*], [*Class list-name*, *Class list-name*]),
([*Class list-name*, *Class list-name*], [*Class list-name*, *Class list-name*]),
([*NT*, *Class list-name*, *Class list-name*], [*Class list-name*, *Class list-name*]),
([*Boolean*, *Class list-name*], [*Class list-name*, *Class list-name*]),

([*Class Object*], [*Class list-name*, *Class list-name*]),
([], [*Class list-name*, *Class list-name*]),
([*Class list-name*], [*Class list-name*, *Class list-name*]),
([*Class list-name*, *Class list-name*], [*Class list-name*, *Class list-name*]),
([], [*Class list-name*, *Class list-name*]),
([*Void*], [*Class list-name*, *Class list-name*]),

```
  (                    [Class list-name], [Class list-name, Class list-name]),
  (    [Class list-name, Class list-name], [Class list-name, Class list-name]),
  (                              [Void], [Class list-name, Class list-name])]
```

The next definition and three proof rules implement an algorithm to enumarate natural numbers. The command *apply* (*elim pc-end pc-next pc-0* transforms a goal of the form

$pc < n \Longrightarrow P\ pc$

into a series of goals

$P\ (0::'a)$

$P\ (Suc\ 0)$

...

$P\ n$

```
definition intervall :: nat ⇒ nat ⇒ nat ⇒ bool (- ∈ [-, -'))
where
  x ∈ [a, b) ≡ a ≤ x ∧ x < b

lemma pc-0: x < n ⟹ (x ∈ [0, n) ⟹ P x) ⟹ P x
  by (simp add: intervall-def)

lemma pc-next: x ∈ [n0, n) ⟹ P n0 ⟹ (x ∈ [Suc n0, n) ⟹ P x) ⟹ P x
lemma pc-end: x ∈ [n,n) ⟹ P x
  by (unfold intervall-def) arith


lemma types-append [simp]: check-types E 3 (Suc (Suc 0)) (map OK φa)
lemma wt-append [simp]:
  wt-method E list-name [Class list-name] Void 3 0 append-ins
          [(Suc 0, 2, NullPointer, 7, 0)] φa
```

Some abbreviations for readability

```
abbreviation Clist == Class list-name
abbreviation Ctest == Class test-name

definition phi-makelist :: tym (φm)
where
  φm ≡ map (λ(x,y). Some (x, y)) [
    (                              [], [OK Ctest, Err     , Err     ]),
    (                         [Clist], [OK Ctest, Err     , Err     ]),
    (                              [], [OK Clist, Err     , Err     ]),
    (                         [Clist], [OK Clist, Err     , Err     ]),
    (                [Integer, Clist], [OK Clist, Err     , Err     ]),

    (                              [], [OK Clist, Err     , Err     ]),
    (                         [Clist], [OK Clist, Err     , Err     ]),
    (                              [], [OK Clist, OK Clist, Err     ]),
    (                         [Clist], [OK Clist, OK Clist, Err     ]),
    (                [Integer, Clist], [OK Clist, OK Clist, Err     ]),
```

([], [*OK Clist*, *OK Clist*, *Err*]),
([*Clist*], [*OK Clist*, *OK Clist*, *Err*]),
([], [*OK Clist*, *OK Clist*, *OK Clist*]),
([*Clist*], [*OK Clist*, *OK Clist*, *OK Clist*]),
([*Integer*, *Clist*], [*OK Clist*, *OK Clist*, *OK Clist*]),

([], [*OK Clist*, *OK Clist*, *OK Clist*]),
([*Clist*], [*OK Clist*, *OK Clist*, *OK Clist*]),
([*Clist*, *Clist*], [*OK Clist*, *OK Clist*, *OK Clist*]),
([*Void*], [*OK Clist*, *OK Clist*, *OK Clist*]),
([], [*OK Clist*, *OK Clist*, *OK Clist*]),
([*Clist*], [*OK Clist*, *OK Clist*, *OK Clist*]),
([*Clist*, *Clist*], [*OK Clist*, *OK Clist*, *OK Clist*]),
([*Void*], [*OK Clist*, *OK Clist*, *OK Clist*])]

lemma *types-makelist* [*simp*]: *check-types E 3* (*Suc* (*Suc* (*Suc 0*))) (*map OK* φ_m)
lemma *wt-makelist* [*simp*]:
wt-method E test-name [] *Void 3 2 make-list-ins* [] φ_m
lemma *wf-md'E*:
⟦ *wf-prog wf-md P*;
⋀*C S fs ms m*.⟦(*C*,*S*,*fs*,*ms*) ∈ *set P*; *m* ∈ *set ms*⟧ ⟹ *wf-md' P C m* ⟧
⟹ *wf-prog wf-md' P*

The whole program is welltyped:

definition *Phi* :: ty_P (Φ)
where
Φ *C mn* ≡ *if C* = *test-name* ∧ *mn* = *makelist-name then* φ_m *else*
if C = *list-name* ∧ *mn* = *append-name then* φ_a *else* []

lemma *wf-prog*:
wf-jvm-prog$_\Phi$ *E*

4.26.4 Conformance

Execution of the program will be typesafe, because its start state conforms to the welltyping:

lemma *E*,Φ ⊢ *start-state E test-name makelist-name* √

4.26.5 Example for code generation: inferring method types

definition *test-kil* :: *jvm-prog* ⇒ *cname* ⇒ *ty list* ⇒ *ty* ⇒ *nat* ⇒ *nat* ⇒
ex-table ⇒ *instr list* ⇒ ty_i' *err list*
where
test-kil G C pTs rT mxs mxl et instr ≡
(*let first* = *Some* ([],(*OK* (*Class C*))#(*map OK pTs*)@(*replicate mxl Err*));
start = *OK first*#(*replicate* (*size instr* − *1*) (*OK None*))
in kiljvm G mxs (*1*+*size pTs*+*mxl*) *rT instr et start*)

lemma [*code*]:
unstables r step ss =
fold (λ*p A*. *if* ¬*stable r step ss p then insert p A else A*) [*0*..<*size ss*] {}
proof −
have *unstables r step ss* = (*UN p*:{..<*size ss*}. *if* ¬*stable r step ss p then* {*p*} *else* {})
apply (*unfold unstables-def*)

```
    apply (rule equalityI)
    apply (rule subsetI)
    apply (erule CollectE)
    apply (erule conjE)
    apply (rule UN-I)
    apply simp
    apply simp
    apply (rule subsetI)
    apply (erule UN-E)
    apply (case-tac ¬ stable r step ss p)
    apply simp+
    done
  also have ⋀f. (UN p:{..<size ss}. f p) = Union (set (map f [0..<size ss])) by auto
  also note Sup-set-fold also note fold-map
  also have (∪) ∘ (λp. if ¬ stable r step ss p then {p} else {}) =
          (λp A. if ¬stable r step ss p then insert p A else A)
    by(auto simp add: fun-eq-iff)
  finally show ?thesis .
qed

definition some-elem :: 'a set ⇒ 'a where [code del]:
  some-elem = (%S. SOME x. x : S)
code-printing
  constant some-elem ⇀ (SML) (case/ - of/ Set/ xs/ =>/ hd/ xs)
```

This code setup is just a demonstration and *not* sound!

```
notepad begin
  have some-elem (set [False, True]) = False by eval
  moreover have some-elem (set [True, False]) = True by eval
  ultimately have False by (simp add: some-elem-def)
end

lemma [code]:
  iter f step ss w = while (λ(ss, w). ¬ Set.is-empty w)
    (λ(ss, w).
       let p = some-elem w in propa f (step p (ss ! p)) ss (w − {p}))
    (ss, w)
  unfolding iter-def Set.is-empty-def some-elem-def ..

lemma JVM-sup-unfold [code]:
 JVM-SemiType.sup S m n = lift2 (Opt.sup
      (Product.sup (Listn.sup (SemiType.sup S))
        (λx y. OK (map2 (lift2 (SemiType.sup S)) x y))))
  apply (unfold JVM-SemiType.sup-def JVM-SemiType.sl-def Opt.esl-def Err.sl-def
         stk-esl-def loc-sl-def Product.esl-def
         Listn.sl-def upto-esl-def SemiType.esl-def Err.esl-def)
  by simp

lemmas [code] = SemiType.sup-def [unfolded exec-lub-def] JVM-le-unfold

lemmas [code] = lesub-def plussub-def

lemma [code]:
  is-refT T = (case T of NT ⇒ True | Class C ⇒ True | - ⇒ False)
```

by (*simp add*: *is-refT-def split*: *ty.split*)

declare *app$_i$.simps* [*code*]

lemma [*code*]:
app$_i$ (*Getfield F C*, *P*, *pc*, *mxs*, *T$_r$*, (*T*#*ST*, *LT*)) =
Predicate.holds (*Predicate.bind* (*sees-field-i-i-i-o-i P C F C*) (λ*T$_f$*. *if P* ⊢ *T* ≤ *Class C then Predicate.single* () *else bot*))
by(*auto simp add*: *Predicate.holds-eq intro*: *sees-field-i-i-i-o-iI elim*: *sees-field-i-i-i-o-iE*)

lemma [*code*]:
app$_i$ (*Putfield F C*, *P*, *pc*, *mxs*, *T$_r$*, (*T$_1$*#*T$_2$*#*ST*, *LT*)) =
Predicate.holds (*Predicate.bind* (*sees-field-i-i-i-o-i P C F C*) (λ*T$_f$*. *if P* ⊢ *T$_2$* ≤ (*Class C*) ∧ *P* ⊢ *T$_1$* ≤ *T$_f$ then Predicate.single* () *else bot*))
by(*auto simp add*: *Predicate.holds-eq simp del*: *eval-bind split*: *if-split-asm elim*!: *sees-field-i-i-i-o-iE Predicate.bindE intro*: *Predicate.bindI sees-field-i-i-i-o-iI*)

lemma [*code*]:
app$_i$ (*Invoke M n*, *P*, *pc*, *mxs*, *T$_r$*, (*ST*,*LT*)) =
(*n* < *length ST* ∧
(*ST*!*n* ≠ *NT* ⟶
(*case ST*!*n of*
Class C ⇒ *Predicate.holds* (*Predicate.bind* (*Method-i-i-i-o-o-o-o P C M*) (λ(*Ts*, *T*, *m*, *D*). *if P* ⊢ *rev* (*take n ST*) [≤] *Ts then Predicate.single* () *else bot*))
| - ⇒ *False*)))
by (*fastforce simp add*: *Predicate.holds-eq simp del*: *eval-bind split*: *ty.split-asm if-split-asm intro*: *bindI Method-i-i-i-o-o-o-oI elim*!: *bindE Method-i-i-i-o-o-o-oE*)

lemmas [*code*] =
SemiType.sup-def [*unfolded exec-lub-def*]
widen.equation
is-relevant-class.simps

definition *test1* **where**
test1 = *test-kil E list-name* [*Class list-name*] *Void 3 0*
[(*Suc 0*, *2*, *NullPointer*, *7*, *0*)] *append-ins*
definition *test2* **where**
test2 = *test-kil E test-name* [] *Void 3 2* [] *make-list-ins*
definition *test3* **where** *test3* = φ_a
definition *test4* **where** *test4* = φ_m

ML-val ‹
if @{*code test1*} = @{*code map*} @{*code OK*} @{*code test3*} *then* () *else error wrong result*;
if @{*code test2*} = @{*code map*} @{*code OK*} @{*code test4*} *then* () *else error wrong result*
›

end

Chapter 5

Compilation

5.1 An Intermediate Language

theory *J1* **imports** *../J/BigStep* **begin**

type-synonym $expr_1 = nat\ exp$
type-synonym $J_1\text{-}prog = expr_1\ prog$
type-synonym $state_1 = heap \times (val\ list)$

primrec
  max-vars :: *'a exp* ⇒ *nat*
  and *max-varss* :: *'a exp list* ⇒ *nat*
where
  max-vars(*new C*) = *0*
| *max-vars*(*Cast C e*) = *max-vars e*
| *max-vars*(*Val v*) = *0*
| *max-vars*(e_1 «*bop*» e_2) = *max* (*max-vars* e_1) (*max-vars* e_2)
| *max-vars*(*Var V*) = *0*
| *max-vars*(*V*:=*e*) = *max-vars e*
| *max-vars*(*e*·*F*{*D*}) = *max-vars e*
| *max-vars*(*FAss* e_1 *F D* e_2) = *max* (*max-vars* e_1) (*max-vars* e_2)
| *max-vars*(*e*·*M*(*es*)) = *max* (*max-vars e*) (*max-varss es*)
| *max-vars*({*V*:*T*; *e*}) = *max-vars e* + *1*
| *max-vars*(e_1;;e_2) = *max* (*max-vars* e_1) (*max-vars* e_2)
| *max-vars*(*if* (*e*) e_1 *else* e_2) =
   max (*max-vars e*) (*max* (*max-vars* e_1) (*max-vars* e_2))
| *max-vars*(*while* (*b*) *e*) = *max* (*max-vars b*) (*max-vars e*)
| *max-vars*(*throw e*) = *max-vars e*
| *max-vars*(*try* e_1 *catch*(*C V*) e_2) = *max* (*max-vars* e_1) (*max-vars* e_2 + *1*)

| *max-varss* [] = *0*
| *max-varss* (*e*#*es*) = *max* (*max-vars e*) (*max-varss es*)

inductive
  $eval_1$ :: $J_1\text{-}prog \Rightarrow expr_1 \Rightarrow state_1 \Rightarrow expr_1 \Rightarrow state_1 \Rightarrow bool$
        (- ⊢$_1$ ((*1*⟨-,/-⟩) ⇒/ (*1*⟨-,/-⟩)) [*51*,*0*,*0*,*0*,*0*] *81*)
  and $evals_1$ :: $J_1\text{-}prog \Rightarrow expr_1\ list \Rightarrow state_1 \Rightarrow expr_1\ list \Rightarrow state_1 \Rightarrow bool$
         (- ⊢$_1$ ((*1*⟨-,/-⟩) [⇒]/ (*1*⟨-,/-⟩)) [*51*,*0*,*0*,*0*,*0*] *81*)
  for P :: $J_1\text{-}prog$
where

New_1:
$[\![$ *new-Addr* h = *Some* a; $P \vdash C$ *has-fields FDTs*; $h' = h(a \mapsto (C, \textit{init-fields FDTs}))$ $]\!]$
$\Longrightarrow P \vdash_1 \langle \textit{new } C, (h,l)\rangle \Rightarrow \langle \textit{addr } a, (h',l)\rangle$
| $NewFail_1$:
new-Addr h = *None* $\Longrightarrow$
$P \vdash_1 \langle \textit{new } C, (h,l)\rangle \Rightarrow \langle \textit{THROW OutOfMemory}, (h,l)\rangle$

| $Cast_1$:
$[\![P \vdash_1 \langle e, s_0\rangle \Rightarrow \langle \textit{addr } a, (h,l)\rangle;\ h\ a = \textit{Some}(D,fs);\ P \vdash D \preceq^* C]\!]$
$\Longrightarrow P \vdash_1 \langle \textit{Cast } C\ e, s_0\rangle \Rightarrow \langle \textit{addr } a, (h,l)\rangle$
| $CastNull_1$:
$P \vdash_1 \langle e, s_0\rangle \Rightarrow \langle \textit{null}, s_1\rangle \Longrightarrow$
$P \vdash_1 \langle \textit{Cast } C\ e, s_0\rangle \Rightarrow \langle \textit{null}, s_1\rangle$
| $CastFail_1$:
$[\![P \vdash_1 \langle e, s_0\rangle \Rightarrow \langle \textit{addr } a, (h,l)\rangle;\ h\ a = \textit{Some}(D,fs);\ \neg\ P \vdash D \preceq^* C]\!]$
$\Longrightarrow P \vdash_1 \langle \textit{Cast } C\ e, s_0\rangle \Rightarrow \langle \textit{THROW ClassCast}, (h,l)\rangle$
| $CastThrow_1$:
$P \vdash_1 \langle e, s_0\rangle \Rightarrow \langle \textit{throw } e', s_1\rangle \Longrightarrow$
$P \vdash_1 \langle \textit{Cast } C\ e, s_0\rangle \Rightarrow \langle \textit{throw } e', s_1\rangle$

| Val_1:
$P \vdash_1 \langle \textit{Val } v, s\rangle \Rightarrow \langle \textit{Val } v, s\rangle$

| $BinOp_1$:
$[\![P \vdash_1 \langle e_1, s_0\rangle \Rightarrow \langle \textit{Val } v_1, s_1\rangle;\ P \vdash_1 \langle e_2, s_1\rangle \Rightarrow \langle \textit{Val } v_2, s_2\rangle;\ \textit{binop}(bop, v_1, v_2) = \textit{Some } v]\!]$
$\Longrightarrow P \vdash_1 \langle e_1\ \textit{«bop»}\ e_2, s_0\rangle \Rightarrow \langle \textit{Val } v, s_2\rangle$
| $BinOpThrow_{11}$:
$P \vdash_1 \langle e_1, s_0\rangle \Rightarrow \langle \textit{throw } e, s_1\rangle \Longrightarrow$
$P \vdash_1 \langle e_1\ \textit{«bop»}\ e_2,\ s_0\rangle \Rightarrow \langle \textit{throw } e, s_1\rangle$
| $BinOpThrow_{21}$:
$[\![P \vdash_1 \langle e_1, s_0\rangle \Rightarrow \langle \textit{Val } v_1, s_1\rangle;\ P \vdash_1 \langle e_2, s_1\rangle \Rightarrow \langle \textit{throw } e, s_2\rangle]\!]$
$\Longrightarrow P \vdash_1 \langle e_1\ \textit{«bop»}\ e_2, s_0\rangle \Rightarrow \langle \textit{throw } e, s_2\rangle$

| Var_1:
$[\![ls!i = v;\ i < \textit{size } ls]\!] \Longrightarrow$
$P \vdash_1 \langle \textit{Var } i, (h,ls)\rangle \Rightarrow \langle \textit{Val } v, (h,ls)\rangle$

| $LAss_1$:
$[\![P \vdash_1 \langle e, s_0\rangle \Rightarrow \langle \textit{Val } v, (h,ls)\rangle;\ i < \textit{size } ls;\ ls' = ls[i := v]]\!]$
$\Longrightarrow P \vdash_1 \langle i := e, s_0\rangle \Rightarrow \langle \textit{unit}, (h,ls')\rangle$
| $LAssThrow_1$:
$P \vdash_1 \langle e, s_0\rangle \Rightarrow \langle \textit{throw } e', s_1\rangle \Longrightarrow$
$P \vdash_1 \langle i := e, s_0\rangle \Rightarrow \langle \textit{throw } e', s_1\rangle$

| $FAcc_1$:
$[\![P \vdash_1 \langle e, s_0\rangle \Rightarrow \langle \textit{addr } a, (h,ls)\rangle;\ h\ a = \textit{Some}(C,fs);\ fs(F,D) = \textit{Some } v]\!]$
$\Longrightarrow P \vdash_1 \langle e{\cdot}F\{D\}, s_0\rangle \Rightarrow \langle \textit{Val } v, (h,ls)\rangle$
| $FAccNull_1$:
$P \vdash_1 \langle e, s_0\rangle \Rightarrow \langle \textit{null}, s_1\rangle \Longrightarrow$
$P \vdash_1 \langle e{\cdot}F\{D\}, s_0\rangle \Rightarrow \langle \textit{THROW NullPointer}, s_1\rangle$
| $FAccThrow_1$:
$P \vdash_1 \langle e, s_0\rangle \Rightarrow \langle \textit{throw } e', s_1\rangle \Longrightarrow$
$P \vdash_1 \langle e{\cdot}F\{D\}, s_0\rangle \Rightarrow \langle \textit{throw } e', s_1\rangle$

| $FAss_1$:
$[\![P \vdash_1 \langle e_1,s_0\rangle \Rightarrow \langle addr\ a,s_1\rangle;\ P \vdash_1 \langle e_2,s_1\rangle \Rightarrow \langle Val\ v,(h_2,l_2)\rangle;$
$h_2\ a = Some(C,fs);\ fs' = fs((F,D)\mapsto v);\ h_2' = h_2(a\mapsto(C,fs'))\ ]\!]$
$\Longrightarrow P \vdash_1 \langle e_1{\cdot}F\{D\}:= e_2,s_0\rangle \Rightarrow \langle unit,(h_2',l_2)\rangle$
| $FAssNull_1$:
$[\![P \vdash_1 \langle e_1,s_0\rangle \Rightarrow \langle null,s_1\rangle;\ \ P \vdash_1 \langle e_2,s_1\rangle \Rightarrow \langle Val\ v,s_2\rangle\ ]\!]$
$\Longrightarrow P \vdash_1 \langle e_1{\cdot}F\{D\}:= e_2,s_0\rangle \Rightarrow \langle THROW\ NullPointer,s_2\rangle$
| $FAssThrow_{11}$:
$P \vdash_1 \langle e_1,s_0\rangle \Rightarrow \langle throw\ e',s_1\rangle \Longrightarrow$
$P \vdash_1 \langle e_1{\cdot}F\{D\}:= e_2,s_0\rangle \Rightarrow \langle throw\ e',s_1\rangle$
| $FAssThrow_{21}$:
$[\![P \vdash_1 \langle e_1,s_0\rangle \Rightarrow \langle Val\ v,s_1\rangle;\ P \vdash_1 \langle e_2,s_1\rangle \Rightarrow \langle throw\ e',s_2\rangle\ ]\!]$
$\Longrightarrow P \vdash_1 \langle e_1{\cdot}F\{D\}:= e_2,s_0\rangle \Rightarrow \langle throw\ e',s_2\rangle$

| $CallObjThrow_1$:
$P \vdash_1 \langle e,s_0\rangle \Rightarrow \langle throw\ e',s_1\rangle \Longrightarrow$
$P \vdash_1 \langle e{\cdot}M(es),s_0\rangle \Rightarrow \langle throw\ e',s_1\rangle$
| $CallNull_1$:
$[\![P \vdash_1 \langle e,s_0\rangle \Rightarrow \langle null,s_1\rangle;\ P \vdash_1 \langle es,s_1\rangle\ [\Rightarrow]\ \langle map\ Val\ vs,s_2\rangle\ ]\!]$
$\Longrightarrow P \vdash_1 \langle e{\cdot}M(es),s_0\rangle \Rightarrow \langle THROW\ NullPointer,s_2\rangle$
| $Call_1$:
$[\![P \vdash_1 \langle e,s_0\rangle \Rightarrow \langle addr\ a,s_1\rangle;\ P \vdash_1 \langle es,s_1\rangle\ [\Rightarrow]\ \langle map\ Val\ vs,(h_2,ls_2)\rangle;$
$h_2\ a = Some(C,fs);\ P \vdash C\ sees\ M{:}Ts\rightarrow T = body\ in\ D;$
$size\ vs = size\ Ts;\ ls_2' = (Addr\ a)\ \#\ vs\ @\ replicate\ (max\text{-}vars\ body)\ undefined;$
$P \vdash_1 \langle body,(h_2,ls_2')\rangle \Rightarrow \langle e',(h_3,ls_3)\rangle\ ]\!]$
$\Longrightarrow P \vdash_1 \langle e{\cdot}M(es),s_0\rangle \Rightarrow \langle e',(h_3,ls_2)\rangle$
| $CallParamsThrow_1$:
$[\![P \vdash_1 \langle e,s_0\rangle \Rightarrow \langle Val\ v,s_1\rangle;\ P \vdash_1 \langle es,s_1\rangle\ [\Rightarrow]\ \langle es',s_2\rangle;$
$es' = map\ Val\ vs\ @\ throw\ ex\ \#\ es_2\ ]\!]$
$\Longrightarrow P \vdash_1 \langle e{\cdot}M(es),s_0\rangle \Rightarrow \langle throw\ ex,s_2\rangle$

| $Block_1$:
$P \vdash_1 \langle e,s_0\rangle \Rightarrow \langle e',s_1\rangle \Longrightarrow P \vdash_1 \langle Block\ i\ T\ e,s_0\rangle \Rightarrow \langle e',s_1\rangle$

| Seq_1:
$[\![P \vdash_1 \langle e_0,s_0\rangle \Rightarrow \langle Val\ v,s_1\rangle;\ P \vdash_1 \langle e_1,s_1\rangle \Rightarrow \langle e_2,s_2\rangle\ ]\!]$
$\Longrightarrow P \vdash_1 \langle e_0;;e_1,s_0\rangle \Rightarrow \langle e_2,s_2\rangle$
| $SeqThrow_1$:
$P \vdash_1 \langle e_0,s_0\rangle \Rightarrow \langle throw\ e,s_1\rangle \Longrightarrow$
$P \vdash_1 \langle e_0;;e_1,s_0\rangle \Rightarrow \langle throw\ e,s_1\rangle$

| $CondT_1$:
$[\![P \vdash_1 \langle e,s_0\rangle \Rightarrow \langle true,s_1\rangle;\ P \vdash_1 \langle e_1,s_1\rangle \Rightarrow \langle e',s_2\rangle\ ]\!]$
$\Longrightarrow P \vdash_1 \langle if\ (e)\ e_1\ else\ e_2,s_0\rangle \Rightarrow \langle e',s_2\rangle$
| $CondF_1$:
$[\![P \vdash_1 \langle e,s_0\rangle \Rightarrow \langle false,s_1\rangle;\ P \vdash_1 \langle e_2,s_1\rangle \Rightarrow \langle e',s_2\rangle\ ]\!]$
$\Longrightarrow P \vdash_1 \langle if\ (e)\ e_1\ else\ e_2,s_0\rangle \Rightarrow \langle e',s_2\rangle$
| $CondThrow_1$:
$P \vdash_1 \langle e,s_0\rangle \Rightarrow \langle throw\ e',s_1\rangle \Longrightarrow$
$P \vdash_1 \langle if\ (e)\ e_1\ else\ e_2,\ s_0\rangle \Rightarrow \langle throw\ e',s_1\rangle$

| $WhileF_1$:
$P \vdash_1 \langle e,s_0\rangle \Rightarrow \langle false,s_1\rangle \Longrightarrow$

$P \vdash_1 \langle \mathit{while}\ (e)\ c, s_0\rangle \Rightarrow \langle \mathit{unit}, s_1\rangle$
| *WhileT*$_1$:
$[\![\ P \vdash_1 \langle e, s_0\rangle \Rightarrow \langle \mathit{true}, s_1\rangle;\ P \vdash_1 \langle c, s_1\rangle \Rightarrow \langle \mathit{Val}\ v_1, s_2\rangle;$
$P \vdash_1 \langle \mathit{while}\ (e)\ c, s_2\rangle \Rightarrow \langle e_3, s_3\rangle\ ]\!]$
$\Longrightarrow P \vdash_1 \langle \mathit{while}\ (e)\ c, s_0\rangle \Rightarrow \langle e_3, s_3\rangle$
| *WhileCondThrow*$_1$:
$P \vdash_1 \langle e, s_0\rangle \Rightarrow \langle \mathit{throw}\ e', s_1\rangle \Longrightarrow$
$P \vdash_1 \langle \mathit{while}\ (e)\ c, s_0\rangle \Rightarrow \langle \mathit{throw}\ e', s_1\rangle$
| *WhileBodyThrow*$_1$:
$[\![\ P \vdash_1 \langle e, s_0\rangle \Rightarrow \langle \mathit{true}, s_1\rangle;\ P \vdash_1 \langle c, s_1\rangle \Rightarrow \langle \mathit{throw}\ e', s_2\rangle]\!]$
$\Longrightarrow P \vdash_1 \langle \mathit{while}\ (e)\ c, s_0\rangle \Rightarrow \langle \mathit{throw}\ e', s_2\rangle$

| *Throw*$_1$:
$P \vdash_1 \langle e, s_0\rangle \Rightarrow \langle \mathit{addr}\ a, s_1\rangle \Longrightarrow$
$P \vdash_1 \langle \mathit{throw}\ e, s_0\rangle \Rightarrow \langle \mathit{Throw}\ a, s_1\rangle$
| *ThrowNull*$_1$:
$P \vdash_1 \langle e, s_0\rangle \Rightarrow \langle \mathit{null}, s_1\rangle \Longrightarrow$
$P \vdash_1 \langle \mathit{throw}\ e, s_0\rangle \Rightarrow \langle \mathit{THROW}\ \mathit{NullPointer}, s_1\rangle$
| *ThrowThrow*$_1$:
$P \vdash_1 \langle e, s_0\rangle \Rightarrow \langle \mathit{throw}\ e', s_1\rangle \Longrightarrow$
$P \vdash_1 \langle \mathit{throw}\ e, s_0\rangle \Rightarrow \langle \mathit{throw}\ e', s_1\rangle$

| *Try*$_1$:
$P \vdash_1 \langle e_1, s_0\rangle \Rightarrow \langle \mathit{Val}\ v_1, s_1\rangle \Longrightarrow$
$P \vdash_1 \langle \mathit{try}\ e_1\ \mathit{catch}(C\ i)\ e_2, s_0\rangle \Rightarrow \langle \mathit{Val}\ v_1, s_1\rangle$
| *TryCatch*$_1$:
$[\![\ P \vdash_1 \langle e_1, s_0\rangle \Rightarrow \langle \mathit{Throw}\ a, (h_1, ls_1)\rangle;$
$h_1\ a = \mathit{Some}(D, fs);\ P \vdash D \preceq^* C;\ i < \mathit{length}\ ls_1;$
$P \vdash_1 \langle e_2, (h_1, ls_1[i := \mathit{Addr}\ a])\rangle \Rightarrow \langle e_2', (h_2, ls_2)\rangle\ ]\!]$
$\Longrightarrow P \vdash_1 \langle \mathit{try}\ e_1\ \mathit{catch}(C\ i)\ e_2, s_0\rangle \Rightarrow \langle e_2', (h_2, ls_2)\rangle$
| *TryThrow*$_1$:
$[\![\ P \vdash_1 \langle e_1, s_0\rangle \Rightarrow \langle \mathit{Throw}\ a, (h_1, ls_1)\rangle;\ h_1\ a = \mathit{Some}(D, fs);\ \neg\ P \vdash D \preceq^* C\ ]\!]$
$\Longrightarrow P \vdash_1 \langle \mathit{try}\ e_1\ \mathit{catch}(C\ i)\ e_2, s_0\rangle \Rightarrow \langle \mathit{Throw}\ a, (h_1, ls_1)\rangle$

| *Nil*$_1$:
$P \vdash_1 \langle [], s\rangle\ [\Rightarrow]\ \langle [], s\rangle$

| *Cons*$_1$:
$[\![\ P \vdash_1 \langle e, s_0\rangle \Rightarrow \langle \mathit{Val}\ v, s_1\rangle;\ P \vdash_1 \langle es, s_1\rangle\ [\Rightarrow]\ \langle es', s_2\rangle\ ]\!]$
$\Longrightarrow P \vdash_1 \langle e \# es, s_0\rangle\ [\Rightarrow]\ \langle \mathit{Val}\ v\ \#\ es', s_2\rangle$
| *ConsThrow*$_1$:
$P \vdash_1 \langle e, s_0\rangle \Rightarrow \langle \mathit{throw}\ e', s_1\rangle \Longrightarrow$
$P \vdash_1 \langle e \# es, s_0\rangle\ [\Rightarrow]\ \langle \mathit{throw}\ e'\ \#\ es,\ s_1\rangle$

lemma *eval*$_1$*-preserves-len*:
$P \vdash_1 \langle e_0, (h_0, ls_0)\rangle \Rightarrow \langle e_1, (h_1, ls_1)\rangle \Longrightarrow \mathit{length}\ ls_0 = \mathit{length}\ ls_1$
and *evals*$_1$*-preserves-len*:
$P \vdash_1 \langle es_0, (h_0, ls_0)\rangle\ [\Rightarrow]\ \langle es_1, (h_1, ls_1)\rangle \Longrightarrow \mathit{length}\ ls_0 = \mathit{length}\ ls_1$

lemma *evals*$_1$*-preserves-elen*:
$\bigwedge es'\ s\ s'.\ P \vdash_1 \langle es, s\rangle\ [\Rightarrow]\ \langle es', s'\rangle \Longrightarrow \mathit{length}\ es = \mathit{length}\ es'$

lemma *eval*$_1$*-final*: $P \vdash_1 \langle e, s\rangle \Rightarrow \langle e', s'\rangle \Longrightarrow \mathit{final}\ e'$
and *evals*$_1$*-final*: $P \vdash_1 \langle es, s\rangle\ [\Rightarrow]\ \langle es', s'\rangle \Longrightarrow \mathit{finals}\ es'$

end

5.2 Well-Formedness of Intermediate Language

theory *J1WellForm*
imports *../J/JWellForm J1*
begin

5.2.1 Well-Typedness

type-synonym
env$_1$ = *ty list* — type environment indexed by variable number

inductive
WT$_1$ :: [*J*$_1$-*prog*,*env*$_1$, *expr*$_1$, *ty*] ⇒ *bool*
((-,- ⊢$_1$/ - :: -) [*51*,*51*,*51*]*50*)
and *WTs*$_1$ :: [*J*$_1$-*prog*,*env*$_1$, *expr*$_1$ *list*, *ty list*] ⇒ *bool*
((-,- ⊢$_1$/ - [::] -) [*51*,*51*,*51*]*50*)
for *P* :: *J*$_1$-*prog*
where

WTNew$_1$:
is-class P C ⟹
P,*E* ⊢$_1$ *new C* :: *Class C*

| *WTCast*$_1$:
⟦ *P*,*E* ⊢$_1$ *e* :: *Class D*; *is-class P C*; *P* ⊢ *C* ≼* *D* ∨ *P* ⊢ *D* ≼* *C* ⟧
⟹ *P*,*E* ⊢$_1$ *Cast C e* :: *Class C*

| *WTVal*$_1$:
typeof v = *Some T* ⟹
P,*E* ⊢$_1$ *Val v* :: *T*

| *WTVar*$_1$:
⟦ *E*!*i* = *T*; *i* < *size E* ⟧
⟹ *P*,*E* ⊢$_1$ *Var i* :: *T*

| *WTBinOp*$_1$:
⟦ *P*,*E* ⊢$_1$ *e*$_1$ :: *T*$_1$; *P*,*E* ⊢$_1$ *e*$_2$ :: *T*$_2$;
case bop of Eq ⇒ (*P* ⊢ *T*$_1$ ≤ *T*$_2$ ∨ *P* ⊢ *T*$_2$ ≤ *T*$_1$) ∧ *T* = *Boolean*
| *Add* ⇒ *T*$_1$ = *Integer* ∧ *T*$_2$ = *Integer* ∧ *T* = *Integer* ⟧
⟹ *P*,*E* ⊢$_1$ *e*$_1$ «*bop*» *e*$_2$ :: *T*

| *WTLAss*$_1$:
⟦ *E*!*i* = *T*; *i* < *size E*; *P*,*E* ⊢$_1$ *e* :: *T*′; *P* ⊢ *T*′ ≤ *T* ⟧
⟹ *P*,*E* ⊢$_1$ *i*:=*e* :: *Void*

| *WTFAcc*$_1$:
⟦ *P*,*E* ⊢$_1$ *e* :: *Class C*; *P* ⊢ *C sees F*:*T in D* ⟧
⟹ *P*,*E* ⊢$_1$ *e*·*F*{*D*} :: *T*

| *WTFAss*$_1$:
⟦ *P*,*E* ⊢$_1$ *e*$_1$ :: *Class C*; *P* ⊢ *C sees F*:*T in D*; *P*,*E* ⊢$_1$ *e*$_2$ :: *T*′; *P* ⊢ *T*′ ≤ *T* ⟧

$\Longrightarrow P,E \vdash_1 e_1{\cdot}F\{D\} := e_2 :: Void$

| $WTCall_1$:
$[\![P,E \vdash_1 e :: Class\ C;\ P \vdash C\ sees\ M{:}Ts' \to T = m\ in\ D;$
$P,E \vdash_1 es\ [::]\ Ts;\ P \vdash Ts\ [\leq]\ Ts']\!]$
$\Longrightarrow P,E \vdash_1 e{\cdot}M(es) :: T$

| $WTBlock_1$:
$[\![is\text{-}type\ P\ T;\ P,E@[T] \vdash_1 e{::}T']\!]$
$\Longrightarrow P,E \vdash_1 \{i{:}T;\ e\} :: T'$

| $WTSeq_1$:
$[\![P,E \vdash_1 e_1{::}T_1;\ P,E \vdash_1 e_2{::}T_2]\!]$
$\Longrightarrow P,E \vdash_1 e_1;;e_2 :: T_2$

| $WTCond_1$:
$[\![P,E \vdash_1 e :: Boolean;\ P,E \vdash_1 e_1{::}T_1;\ P,E \vdash_1 e_2{::}T_2;$
$P \vdash T_1 \leq T_2 \vee P \vdash T_2 \leq T_1;\ P \vdash T_1 \leq T_2 \longrightarrow T = T_2;\ P \vdash T_2 \leq T_1 \longrightarrow T = T_1]\!]$
$\Longrightarrow P,E \vdash_1 if\ (e)\ e_1\ else\ e_2 :: T$

| $WTWhile_1$:
$[\![P,E \vdash_1 e :: Boolean;\ P,E \vdash_1 c{::}T]\!]$
$\Longrightarrow P,E \vdash_1 while\ (e)\ c :: Void$

| $WTThrow_1$:
$P,E \vdash_1 e :: Class\ C \Longrightarrow$
$P,E \vdash_1 throw\ e :: Void$

| $WTTry_1$:
$[\![P,E \vdash_1 e_1 :: T;\ P,E@[Class\ C] \vdash_1 e_2 :: T;\ is\text{-}class\ P\ C]\!]$
$\Longrightarrow P,E \vdash_1 try\ e_1\ catch(C\ i)\ e_2 :: T$

| $WTNil_1$:
$P,E \vdash_1 [] [::] []$

| $WTCons_1$:
$[\![P,E \vdash_1 e :: T;\ P,E \vdash_1 es\ [::]\ Ts]\!]$
$\Longrightarrow P,E \vdash_1 e\#es\ [::]\ T\#Ts$

lemma WTs_1-*same-size*: $\bigwedge Ts.\ P,E \vdash_1 es\ [::]\ Ts \Longrightarrow size\ es = size\ Ts$

lemma WT_1-*unique*:
$P,E \vdash_1 e :: T_1 \Longrightarrow (\bigwedge T_2.\ P,E \vdash_1 e :: T_2 \Longrightarrow T_1 = T_2)$ **and**
$P,E \vdash_1 es\ [::]\ Ts_1 \Longrightarrow (\bigwedge Ts_2.\ P,E \vdash_1 es\ [::]\ Ts_2 \Longrightarrow Ts_1 = Ts_2)$

lemma assumes *wf*: *wf-prog p P*
shows WT_1-*is-type*: $P,E \vdash_1 e :: T \Longrightarrow set\ E \subseteq types\ P \Longrightarrow is\text{-}type\ P\ T$
and $P,E \vdash_1 es\ [::]\ Ts \Longrightarrow True$

5.2.2 Well-formedness

primrec $\mathcal{B} :: expr_1 \Rightarrow nat \Rightarrow bool$
and $\mathcal{B}s :: expr_1\ list \Rightarrow nat \Rightarrow bool$ **where**
$\mathcal{B}\ (new\ C)\ i = True$ |

$\mathcal{B}\ (Cast\ C\ e)\ i = \mathcal{B}\ e\ i\ |$
$\mathcal{B}\ (Val\ v)\ i = True\ |$
$\mathcal{B}\ (e_1\ «bop»\ e_2)\ i = (\mathcal{B}\ e_1\ i \wedge \mathcal{B}\ e_2\ i)\ |$
$\mathcal{B}\ (Var\ j)\ i = True\ |$
$\mathcal{B}\ (e{\cdot}F\{D\})\ i = \mathcal{B}\ e\ i\ |$
$\mathcal{B}\ (j{:=}e)\ i = \mathcal{B}\ e\ i\ |$
$\mathcal{B}\ (e_1{\cdot}F\{D\} := e_2)\ i = (\mathcal{B}\ e_1\ i \wedge \mathcal{B}\ e_2\ i)\ |$
$\mathcal{B}\ (e{\cdot}M(es))\ i = (\mathcal{B}\ e\ i \wedge \mathcal{B}s\ es\ i)\ |$
$\mathcal{B}\ (\{j{:}T\ ;\ e\})\ i = (i = j \wedge \mathcal{B}\ e\ (i{+}1))\ |$
$\mathcal{B}\ (e_1;;e_2)\ i = (\mathcal{B}\ e_1\ i \wedge \mathcal{B}\ e_2\ i)\ |$
$\mathcal{B}\ (if\ (e)\ e_1\ else\ e_2)\ i = (\mathcal{B}\ e\ i \wedge \mathcal{B}\ e_1\ i \wedge \mathcal{B}\ e_2\ i)\ |$
$\mathcal{B}\ (throw\ e)\ i = \mathcal{B}\ e\ i\ |$
$\mathcal{B}\ (while\ (e)\ c)\ i = (\mathcal{B}\ e\ i \wedge \mathcal{B}\ c\ i)\ |$
$\mathcal{B}\ (try\ e_1\ catch(C\ j)\ e_2)\ i = (\mathcal{B}\ e_1\ i \wedge i{=}j \wedge \mathcal{B}\ e_2\ (i{+}1))\ |$

$\mathcal{B}s\ []\ i = True\ |$
$\mathcal{B}s\ (e\#es)\ i = (\mathcal{B}\ e\ i \wedge \mathcal{B}s\ es\ i)$

definition *wf-J$_1$-mdecl* :: *J$_1$-prog* ⇒ *cname* ⇒ *expr$_1$ mdecl* ⇒ *bool*
where
wf-J$_1$-mdecl P C ≡ λ(*M*,*Ts*,*T*,*body*).
(∃ *T'*. *P*,*Class C*#*Ts* ⊢$_1$ *body* :: *T'* ∧ *P* ⊢ *T'* ≤ *T*) ∧
$\mathcal{D}$ *body* ⌊{..*size Ts*}⌋ ∧ $\mathcal{B}$ *body* (*size Ts* + *1*)

lemma *wf-J$_1$-mdecl*[*simp*]:
wf-J$_1$-mdecl P C (*M*,*Ts*,*T*,*body*) ≡
((∃ *T'*. *P*,*Class C*#*Ts* ⊢$_1$ *body* :: *T'* ∧ *P* ⊢ *T'* ≤ *T*) ∧
$\mathcal{D}$ *body* ⌊{..*size Ts*}⌋ ∧ $\mathcal{B}$ *body* (*size Ts* + *1*))
abbreviation *wf-J$_1$-prog* == *wf-prog wf-J$_1$-mdecl*

end

5.3 Program Compilation

theory *PCompiler*
imports *../Common/WellForm*
begin

definition *compM* :: ('*a* ⇒ '*b*) ⇒ '*a mdecl* ⇒ '*b mdecl*
where
compM f ≡ λ(*M*, *Ts*, *T*, *m*). (*M*, *Ts*, *T*, *f m*)

definition *compC* :: ('*a* ⇒ '*b*) ⇒ '*a cdecl* ⇒ '*b cdecl*
where
compC f ≡ λ(*C*,*D*,*Fdecls*,*Mdecls*). (*C*,*D*,*Fdecls*, *map* (*compM f*) *Mdecls*)

definition *compP* :: ('*a* ⇒ '*b*) ⇒ '*a prog* ⇒ '*b prog*
where
compP f ≡ *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-of (*map* (λ(*x*,*a*,*b*,*c*).(*x*,*a*,*b*,*f c*)) *ts*) =
map-option (λ(*a*,*b*,*c*).(*a*,*b*,*f c*)) ∘ (*map-of ts*)

lemma *class-compP*:
class P C = *Some* (*D*, *fs*, *ms*)
⟹ *class* (*compP f P*) *C* = *Some* (*D*, *fs*, *map* (*compM f*) *ms*)

lemma *class-compPD*:
class (*compP f P*) *C* = *Some* (*D*, *fs*, *cms*)
⟹ ∃ *ms*. *class P C* = *Some*(*D*,*fs*,*ms*) ∧ *cms* = *map* (*compM f*) *ms*

lemma [*simp*]: *is-class* (*compP f P*) *C* = *is-class P C*

lemma [*simp*]: *class* (*compP f P*) *C* = *map-option* (λ*c*. *snd*(*compC f* (*C*,*c*))) (*class P C*)

lemma *sees-methods-compP*:
P ⊢ *C sees-methods Mm* ⟹
compP f P ⊢ *C sees-methods* (*map-option* (λ((*Ts*,*T*,*m*),*D*). ((*Ts*,*T*,*f m*),*D*)) ∘ *Mm*)

lemma *sees-method-compP*:
P ⊢ *C sees M*: *Ts*→*T* = *m in D* ⟹
compP f P ⊢ *C sees M*: *Ts*→*T* = (*f m*) *in D*

lemma [*simp*]:
P ⊢ *C sees M*: *Ts*→*T* = *m in D* ⟹
method (*compP f P*) *C M* = (*D*,*Ts*,*T*,*f m*)

lemma *sees-methods-compPD*:
⟦ *cP* ⊢ *C sees-methods Mm*′; *cP* = *compP f P* ⟧ ⟹
∃ *Mm*. *P* ⊢ *C sees-methods Mm* ∧
Mm′ = (*map-option* (λ((*Ts*,*T*,*m*),*D*). ((*Ts*,*T*,*f m*),*D*)) ∘ *Mm*)

lemma *sees-method-compPD*:
compP f P ⊢ *C sees M*: *Ts*→*T* = *fm in D* ⟹
∃ *m*. *P* ⊢ *C sees M*: *Ts*→*T* = *m in D* ∧ *f m* = *fm*

lemma [*simp*]: *subcls1*(*compP f P*) = *subcls1 P*

lemma *compP-widen*[*simp*]: (*compP f P* ⊢ *T* ≤ *T*′) = (*P* ⊢ *T* ≤ *T*′)

lemma [*simp*]: (*compP f P* ⊢ *Ts* [≤] *Ts*′) = (*P* ⊢ *Ts* [≤] *Ts*′)

lemma [*simp*]: *is-type* (*compP f P*) *T* = *is-type P T*

lemma [*simp*]: (*compP* (*f*::′*a*⇒′*b*) *P* ⊢ *C has-fields FDTs*) = (*P* ⊢ *C has-fields FDTs*)

lemma [*simp*]: *fields* (*compP f P*) *C* = *fields P C*

lemma [*simp*]: (*compP f P* ⊢ *C sees F*:*T in D*) = (*P* ⊢ *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-fst* (*compP f P*) = *distinct-fst P*

lemma [*iff*]: *distinct-fst* (*map* (*compM f*) *ms*) = *distinct-fst ms*

lemma [*iff*]: *wf-syscls* (*compP f P*) = *wf-syscls P*

lemma [*iff*]: *wf-fdecl* (*compP f P*) = *wf-fdecl P*

lemma *set-compP*:
((C,D,*fs*,*ms'*) ∈ *set*(*compP f P*)) =
(∃ *ms*. (C,D,*fs*,*ms*) ∈ *set P* ∧ *ms'* = *map* (*compM f*) *ms*)

lemma *wf-cdecl-compPI*:
⟦ ⋀C M *Ts* T *m*.
⟦ *wf-mdecl* wf_1 P C (M,*Ts*,T,*m*); P ⊢ C *sees* M:*Ts*→T = *m* *in* C ⟧
⟹ *wf-mdecl* wf_2 (*compP f P*) C (M,*Ts*,T, *f m*);
∀ *x*∈*set P*. *wf-cdecl* wf_1 P *x*; *x* ∈ *set* (*compP f P*); *wf-prog p P* ⟧
⟹ *wf-cdecl* wf_2 (*compP f P*) *x*

lemma *wf-prog-compPI*:
assumes *lift*:
⋀C M *Ts* T *m*.
⟦ P ⊢ C *sees* M:*Ts*→T = *m* *in* C; *wf-mdecl* wf_1 P C (M,*Ts*,T,*m*) ⟧
⟹ *wf-mdecl* wf_2 (*compP f P*) C (M,*Ts*,T, *f m*)
and *wf*: *wf-prog* wf_1 P
shows *wf-prog* wf_2 (*compP f P*)

end
theory *Hidden*
imports *List−Index.List-Index*
begin

definition *hidden* :: *'a list* ⇒ *nat* ⇒ *bool* **where**
hidden xs i ≡ *i* < *size xs* ∧ *xs*!*i* ∈ *set*(*drop* (*i*+*1*) *xs*)

lemma *hidden-last-index*: *x* ∈ *set xs* ⟹ *hidden* (*xs* @ [*x*]) (*last-index xs x*)
by(*auto simp add*: *hidden-def nth-append rev-nth*[*symmetric*]
dest: *last-index-less*[*OF - le-refl*])

lemma *hidden-inacc*: *hidden xs i* ⟹ *last-index xs x* ≠ *i*
by(*auto simp add*: *hidden-def last-index-drop last-index-less-size-conv*)

lemma [*simp*]: *hidden xs i* ⟹ *hidden* (*xs*@[*x*]) *i*
by(*auto simp add*:*hidden-def nth-append*)

lemma *fun-upds-apply*:
(*m*(*xs*[↦]*ys*)) *x* =
(*let xs'* = *take* (*size ys*) *xs*
in if x ∈ *set xs' then Some*(*ys* ! *last-index xs' x*) *else m x*)

proof(*induct xs arbitrary*: *m ys*)
case *Nil* **then show** *?case* **by**(*simp add*: *Let-def*)
next
case *Cons* **show** *?case*
proof(*cases ys*)
case *Nil*
then show *?thesis* **by**(*simp add*:*Let-def*)
next
case *Cons'*: *Cons*
then show *?thesis* **using** *Cons* **by**(*simp add*: *Let-def last-index-Cons*)
qed
qed

lemma *map-upds-apply-eq-Some*:
((*m*(*xs*[↦]*ys*)) *x* = *Some y*) =
(*let xs'* = *take* (*size ys*) *xs*
in if x ∈ *set xs' then ys* ! *last-index xs' x* = *y else m x* = *Some y*)
by(*simp add*:*fun-upds-apply Let-def*)

lemma *map-upds-upd-conv-last-index*:
⟦*x* ∈ *set xs*; *size xs* ≤ *size ys* ⟧
⟹ *m*(*xs*[↦]*ys*)(*x*↦*y*) = *m*(*xs*[↦]*ys*[*last-index xs x* := *y*])
by(*rule ext*) (*simp add*:*fun-upds-apply eq-sym-conv Let-def*)

end

5.4 Compilation Stage 1

theory *Compiler1* **imports** *PCompiler J1 Hidden* **begin**

Replacing variable names by indices.

primrec $compE_1$ :: *vname list* ⇒ *expr* ⇒ $expr_1$
and $compEs_1$ :: *vname list* ⇒ *expr list* ⇒ $expr_1$ *list* **where**
$compE_1$ *Vs* (*new C*) = *new C*
| $compE_1$ *Vs* (*Cast C e*) = *Cast C* ($compE_1$ *Vs e*)
| $compE_1$ *Vs* (*Val v*) = *Val v*
| $compE_1$ *Vs* (e_1 «*bop*» e_2) = ($compE_1$ *Vs* e_1) «*bop*» ($compE_1$ *Vs* e_2)
| $compE_1$ *Vs* (*Var V*) = *Var*(*last-index Vs V*)
| $compE_1$ *Vs* (*V*:=*e*) = (*last-index Vs V*):= ($compE_1$ *Vs e*)
| $compE_1$ *Vs* (*e*·*F*{*D*}) = ($compE_1$ *Vs e*)·*F*{*D*}
| $compE_1$ *Vs* (e_1·*F*{*D*}:=e_2) = ($compE_1$ *Vs* e_1)·*F*{*D*} := ($compE_1$ *Vs* e_2)
| $compE_1$ *Vs* (*e*·*M*(*es*)) = ($compE_1$ *Vs e*)·*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) =
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*]: $compEs_1\ Vs\ es = map\ (compE_1\ Vs)\ es$

primrec fin_1:: $expr \Rightarrow expr_1$ **where**
$fin_1(Val\ v) = Val\ v$
| $fin_1(throw\ e) = throw(fin_1\ e)$

lemma *comp-final*: $final\ e \Longrightarrow compE_1\ Vs\ e = fin_1\ e$

lemma [*simp*]:
$\bigwedge Vs.\ max\text{-}vars\ (compE_1\ Vs\ e) = max\text{-}vars\ e$
and $\bigwedge Vs.\ max\text{-}varss\ (compEs_1\ Vs\ es) = max\text{-}varss\ es$

Compiling programs:

definition $compP_1$ :: *J-prog* $\Rightarrow$ J_1*-prog*
where
$compP_1 \equiv compP\ (\lambda(pns,body).\ compE_1\ (this\#pns)\ body)$

end

5.5 Correctness of Stage 1

theory *Correctness1*
imports *J1WellForm Compiler1*
begin

5.5.1 Correctness of program compilation

primrec $unmod$ :: $expr_1 \Rightarrow nat \Rightarrow bool$
and $unmods$ :: $expr_1\ list \Rightarrow nat \Rightarrow bool$ **where**
$unmod\ (new\ C)\ i = True$ |
$unmod\ (Cast\ C\ e)\ i = unmod\ e\ i$ |
$unmod\ (Val\ v)\ i = True$ |
$unmod\ (e_1\ «bop»\ e_2)\ i = (unmod\ e_1\ i \wedge unmod\ e_2\ i)$ |
$unmod\ (Var\ i)\ j = True$ |
$unmod\ (i{:=}e)\ j = (i \neq j \wedge unmod\ e\ j)$ |
$unmod\ (e{\cdot}F\{D\})\ i = unmod\ e\ i$ |
$unmod\ (e_1{\cdot}F\{D\}{:=}e_2)\ i = (unmod\ e_1\ i \wedge unmod\ e_2\ i)$ |
$unmod\ (e{\cdot}M(es))\ i = (unmod\ e\ i \wedge unmods\ es\ i)$ |
$unmod\ \{j{:}T;\ e\}\ i = unmod\ e\ i$ |
$unmod\ (e_1;;e_2)\ i = (unmod\ e_1\ i \wedge\ unmod\ e_2\ i)$ |
$unmod\ (if\ (e)\ e_1\ else\ e_2)\ i = (unmod\ e\ i \wedge unmod\ e_1\ i \wedge unmod\ e_2\ i)$ |
$unmod\ (while\ (e)\ c)\ i = (unmod\ e\ i \wedge unmod\ c\ i)$ |
$unmod\ (throw\ e)\ i = unmod\ e\ i$ |
$unmod\ (try\ e_1\ catch(C\ i)\ e_2)\ j = (unmod\ e_1\ j \wedge (if\ i{=}j\ then\ False\ else\ unmod\ e_2\ j))$ |

$unmods\ ([])\ i = True$ |
$unmods\ (e\#es)\ i = (unmod\ e\ i \wedge unmods\ es\ i)$

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*:
$[\![x \in set\ xs;\ size\ xs \leq size\ ys\ ]\!]$
$\Longrightarrow m_1 \subseteq_m m_2(xs[\mapsto]ys) \Longrightarrow m_1(x{\mapsto}y) \subseteq_m m_2(xs[\mapsto]ys[last\text{-}index\ xs\ x := y])$**lemma** *Block-lem*:
fixes $l :: 'a \rightharpoonup 'b$
assumes *0*: $l \subseteq_m [Vs\ [\mapsto]\ ls]$
and *1*: $l' \subseteq_m [Vs\ [\mapsto]\ ls',\ V{\mapsto}v]$
and *hidden*: $V \in set\ Vs \Longrightarrow ls\,!\,last\text{-}index\ Vs\ V = ls'\,!\,last\text{-}index\ Vs\ V$
and *size*: $size\ ls = size\ ls' \quad size\ Vs < size\ ls'$
shows $l'(V := l\ V) \subseteq_m [Vs\ [\mapsto]\ ls']$

The main theorem:

theorem assumes *wf*: *wwf-J-prog P*
shows *eval*$_1$*-eval*: $P \vdash \langle e,(h,l)\rangle \Rightarrow \langle e',(h',l')\rangle$
$\Longrightarrow (\bigwedge Vs\ ls.\ [\![\ fv\ e \subseteq set\ Vs;\ \ l \subseteq_m [Vs[\mapsto]ls];\ size\ Vs + max\text{-}vars\ e \leq size\ ls\ ]\!]$
$\Longrightarrow \exists ls'.\ compP_1\ P \vdash_1 \langle compE_1\ Vs\ e,(h,ls)\rangle \Rightarrow \langle fin_1\ e',(h',ls')\rangle \wedge l' \subseteq_m [Vs[\mapsto]ls'])$
and *evals*$_1$*-evals*: $P \vdash \langle es,(h,l)\rangle\ [\Rightarrow]\ \langle es',(h',l')\rangle$
$\Longrightarrow (\bigwedge Vs\ ls.\ [\![\ fvs\ es \subseteq set\ Vs;\ \ l \subseteq_m [Vs[\mapsto]ls];\ size\ Vs + max\text{-}varss\ es \leq size\ ls\ ]\!]$
$\Longrightarrow \exists ls'.\ compP_1\ P \vdash_1 \langle compEs_1\ Vs\ es,(h,ls)\rangle\ [\Rightarrow]\ \langle compEs_1\ Vs\ es',(h',ls')\rangle \wedge$
$l' \subseteq_m [Vs[\mapsto]ls'])$

5.5.2 Preservation of well-formedness

The compiler preserves well-formedness. Is less trivial than it may appear. We start with two simple properties: preservation of well-typedness

lemma *compE*$_1$*-pres-wt*: $\bigwedge Vs\ Ts\ U.$
$[\![\ P,[Vs[\mapsto]Ts] \vdash e :: U;\ size\ Ts = size\ Vs\ ]\!]$
$\Longrightarrow compP\ f\ P, Ts \vdash_1 compE_1\ Vs\ e :: U$
and $\bigwedge Vs\ Ts\ Us.$
$[\![\ P,[Vs[\mapsto]Ts] \vdash es\ [::]\ Us;\ size\ Ts = size\ Vs\ ]\!]$
$\Longrightarrow compP\ f\ P, Ts \vdash_1 compEs_1\ Vs\ es\ [::]\ Us$

and the correct block numbering:

lemma $\mathcal{B}$: $\bigwedge Vs\ n.\ size\ Vs = n \Longrightarrow \mathcal{B}\ (compE_1\ Vs\ e)\ n$
and $\mathcal{B}s$: $\bigwedge Vs\ n.\ size\ Vs = n \Longrightarrow \mathcal{B}s\ (compEs_1\ Vs\ es)\ n$

The main complication is preservation of definite assignment $\mathcal{D}$.

lemma *image-last-index*: $A \subseteq set(xs@[x]) \Longrightarrow last\text{-}index\ (xs\ @\ [x])\ `\ A =$
$(if\ x \in A\ then\ insert\ (size\ xs)\ (last\text{-}index\ xs\ `\ (A{-}\{x\}))\ else\ last\text{-}index\ xs\ `\ A)$

lemma *A-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*$_1$:
$\bigwedge A\ Vs.\ [\![\ \mathcal{A}\ e = \lfloor A \rfloor;\ fv\ e \subseteq set\ Vs\ ]\!] \Longrightarrow \mathcal{A}\ (compE_1\ Vs\ e) = \lfloor last\text{-}index\ Vs\ `\ A \rfloor$
and $\bigwedge A\ Vs.\ [\![\ \mathcal{A}s\ es = \lfloor A \rfloor;\ fvs\ es \subseteq set\ Vs\ ]\!] \Longrightarrow \mathcal{A}s\ (compEs_1\ Vs\ es) = \lfloor last\text{-}index\ Vs\ `\ A \rfloor$

lemma *D-None*[*iff*]: $\mathcal{D}\ (e::'a\ exp)\ None$ **and** [*iff*]: $\mathcal{D}s\ (es::'a\ exp\ list)\ None$

lemma *D-last-index-compE*$_1$:
$\bigwedge A\ Vs.\ [\![\ A \subseteq set\ Vs;\ fv\ e \subseteq set\ Vs\ ]\!] \Longrightarrow$
$\mathcal{D}\ e\ \lfloor A \rfloor \Longrightarrow \mathcal{D}\ (compE_1\ Vs\ e)\ \lfloor last\text{-}index\ Vs\ `\ A \rfloor$
and $\bigwedge A\ Vs.\ [\![\ A \subseteq set\ Vs;\ fvs\ es \subseteq set\ Vs\ ]\!] \Longrightarrow$
$\mathcal{D}s\ es\ \lfloor A \rfloor \Longrightarrow \mathcal{D}s\ (compEs_1\ Vs\ es)\ \lfloor last\text{-}index\ Vs\ `\ A \rfloor$

lemma *last-index-image-set*: $distinct\ xs \Longrightarrow last\text{-}index\ xs\ `\ set\ xs = \{..<size\ xs\}$

lemma *D-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-J-prog* $P \Longrightarrow$ *wf-J*$_1$*-prog* $(compP_1\ P)$

end

5.6 Compilation Stage 2

theory *Compiler2*
imports *PCompiler J1* ../*JVM*/*JVMExec*
begin

primrec $compE_2 :: expr_1 \Rightarrow instr\ list$
and $compEs_2 :: expr_1\ list \Rightarrow instr\ list$ **where**
$compE_2\ (new\ C) = [New\ C]$
| $compE_2\ (Cast\ C\ e) = compE_2\ e\ @\ [Checkcast\ C]$
| $compE_2\ (Val\ v) = [Push\ v]$
| $compE_2\ (e_1\ «bop»\ e_2) = compE_2\ e_1\ @\ compE_2\ e_2\ @$
$(case\ bop\ of\ Eq \Rightarrow [CmpEq]$
$\quad | \ Add \Rightarrow [IAdd])$
| $compE_2\ (Var\ i) = [Load\ i]$
| $compE_2\ (i{:=}e) = compE_2\ e\ @\ [Store\ i,\ Push\ Unit]$
| $compE_2\ (e{\cdot}F\{D\}) = compE_2\ e\ @\ [Getfield\ F\ D]$
| $compE_2\ (e_1{\cdot}F\{D\} := e_2) =$
$\quad compE_2\ e_1\ @\ compE_2\ e_2\ @\ [Putfield\ F\ D,\ Push\ Unit]$
| $compE_2\ (e{\cdot}M(es)) = compE_2\ e\ @\ compEs_2\ es\ @\ [Invoke\ M\ (size\ es)]$
| $compE_2\ (\{i{:}T;\ e\}) = compE_2\ e$
| $compE_2\ (e_1;;e_2) = compE_2\ e_1\ @\ [Pop]\ @\ compE_2\ e_2$
| $compE_2\ (if\ (e)\ e_1\ else\ e_2) =$
$\quad (let\ cnd\ \ = compE_2\ e;$
$\quad\ \ thn\ \ = compE_2\ e_1;$
$\quad\ \ els\ \ = compE_2\ e_2;$
$\quad\ \ test\ = IfFalse\ (int(size\ thn + 2));$
$\quad\ \ thnex = Goto\ (int(size\ els + 1))$
$\quad\ in\ cnd\ @\ [test]\ @\ thn\ @\ [thnex]\ @\ els)$
| $compE_2\ (while\ (e)\ c) =$
$\quad (let\ cnd\ \ = compE_2\ e;$
$\quad\ \ bdy\ \ = compE_2\ c;$
$\quad\ \ test\ = IfFalse\ (int(size\ bdy + 3));$
$\quad\ \ loop\ = Goto\ (-int(size\ bdy + size\ cnd + 2))$

```
        in cnd @ [test] @ bdy @ [Pop] @ [loop] @ [Push Unit])
| compE₂ (throw e) = compE₂ e @ [instr.Throw]
| compE₂ (try e₁ catch(C i) e₂) =
   (let catch = compE₂ e₂
    in compE₂ e₁ @ [Goto (int(size catch)+2), Store i] @ catch)

| compEs₂ []     = []
| compEs₂ (e#es) = compE₂ e @ compEs₂ es
```

Compilation of exception table. Is given start address of code to compute absolute addresses necessary in exception table.

```
primrec compxE₂  :: expr₁       ⇒ pc ⇒ nat ⇒ ex-table
 and compxEs₂ :: expr₁ list ⇒ pc ⇒ nat ⇒ ex-table where
  compxE₂ (new C) pc d = []
| compxE₂ (Cast C e) pc d = compxE₂ e pc d
| compxE₂ (Val v) pc d = []
| compxE₂ (e₁ «bop» e₂) pc d =
   compxE₂ e₁ pc d @ compxE₂ e₂ (pc + size(compE₂ e₁)) (d+1)
| compxE₂ (Var i) pc d = []
| compxE₂ (i:=e) pc d = compxE₂ e pc d
| compxE₂ (e·F{D}) pc d = compxE₂ e pc d
| compxE₂ (e₁·F{D} := e₂) pc d =
   compxE₂ e₁ pc d @ compxE₂ e₂ (pc + size(compE₂ e₁)) (d+1)
| compxE₂ (e·M(es)) pc d =
   compxE₂ e pc d @ compxEs₂ es (pc + size(compE₂ e)) (d+1)
| compxE₂ ({i:T; e}) pc d = compxE₂ e pc d
| compxE₂ (e₁;;e₂) pc d =
   compxE₂ e₁ pc d @ compxE₂ e₂ (pc+size(compE₂ e₁)+1) d
| compxE₂ (if (e) e₁ else e₂) pc d =
       (let pc₁   = pc + size(compE₂ e) + 1;
            pc₂   = pc₁ + size(compE₂ e₁) + 1
        in compxE₂ e pc d @ compxE₂ e₁ pc₁ d @ compxE₂ e₂ pc₂ d)
| compxE₂ (while (b) e) pc d =
   compxE₂ b pc d @ compxE₂ e (pc+size(compE₂ b)+1) d
| compxE₂ (throw e) pc d = compxE₂ e pc d
| compxE₂ (try e₁ catch(C i) e₂) pc d =
   (let pc₁ = pc + size(compE₂ e₁)
    in compxE₂ e₁ pc d @ compxE₂ e₂ (pc₁+2) d @ [(pc,pc₁,C,pc₁+1,d)])

| compxEs₂ [] pc d    = []
| compxEs₂ (e#es) pc d = compxE₂ e pc d @ compxEs₂ es (pc+size(compE₂ e)) (d+1)

primrec max-stack :: expr₁ ⇒ nat
 and max-stacks :: expr₁ list ⇒ nat where
  max-stack (new C) = 1
| max-stack (Cast C e) = max-stack e
| max-stack (Val v) = 1
| max-stack (e₁ «bop» e₂) = max (max-stack e₁) (max-stack e₂) + 1
| max-stack (Var i) = 1
| max-stack (i:=e) = max-stack e
| max-stack (e·F{D}) = max-stack e
| max-stack (e₁·F{D} := e₂) = max (max-stack e₁) (max-stack e₂) + 1
| max-stack (e·M(es)) = max (max-stack e) (max-stacks es) + 1
| max-stack ({i:T; e}) = max-stack e
```

| *max-stack* (e_1;;e_2) = *max* (*max-stack* e_1) (*max-stack* e_2)
| *max-stack* (*if* (*e*) e_1 *else* e_2) =
max (*max-stack e*) (*max* (*max-stack* e_1) (*max-stack* e_2))
| *max-stack* (*while* (*e*) *c*) = *max* (*max-stack e*) (*max-stack c*)
| *max-stack* (*throw e*) = *max-stack e*
| *max-stack* (*try* e_1 *catch*(*C i*) e_2) = *max* (*max-stack* e_1) (*max-stack* e_2)

| *max-stacks* [] = *0*
| *max-stacks* (*e*#*es*) = *max* (*max-stack e*) (*1* + *max-stacks es*)

lemma *max-stack1*: *1* ≤ *max-stack e*

definition *compMb*$_2$ :: *expr*$_1$ ⇒ *jvm-method*
where
compMb$_2$ ≡ λ*body*.
let ins = *compE*$_2$ *body* @ [*Return*];
xt = *compxE*$_2$ *body 0 0*
in (*max-stack body*, *max-vars body*, *ins*, *xt*)

definition *compP*$_2$ :: *J*$_1$*-prog* ⇒ *jvm-prog*
where
compP$_2$ ≡ *compP compMb*$_2$

lemma *compMb*$_2$ [*simp*]:
compMb$_2$ *e* = (*max-stack e*, *max-vars e*, *compE*$_2$ *e* @ [*Return*], *compxE*$_2$ *e 0 0*)

end

5.7 Correctness of Stage 2

theory *Correctness2*
imports *HOL−Library.Sublist Compiler2*
begin

5.7.1 Instruction sequences

How to select individual instructions and subsequences of instructions from a program given the class, method and program counter.

definition *before* :: *jvm-prog* ⇒ *cname* ⇒ *mname* ⇒ *nat* ⇒ *instr list* ⇒ *bool*
((-,-,-,-/ ▷ -) [*51*,*0*,*0*,*0*,*51*] *50*) **where**
P,*C*,*M*,*pc* ▷ *is* ⟷ *prefix is* (*drop pc* (*instrs-of P C M*))

definition *at* :: *jvm-prog* ⇒ *cname* ⇒ *mname* ⇒ *nat* ⇒ *instr* ⇒ *bool*
((-,-,-,-/ ▷ -) [*51*,*0*,*0*,*0*,*51*] *50*) **where**
P,*C*,*M*,*pc* ▷ *i* ⟷ (∃ *is*. *drop pc* (*instrs-of P C M*) = *i*#*is*)

lemma [*simp*]: *P*,*C*,*M*,*pc* ▷ []

lemma [*simp*]: *P*,*C*,*M*,*pc* ▷ (*i*#*is*) = (*P*,*C*,*M*,*pc* ▷ *i* ∧ *P*,*C*,*M*,*pc* + *1* ▷ *is*)

lemma [*simp*]: *P*,*C*,*M*,*pc* ▷ (*is*$_1$ @ *is*$_2$) = (*P*,*C*,*M*,*pc* ▷ *is*$_1$ ∧ *P*,*C*,*M*,*pc* + *size is*$_1$ ▷ *is*$_2$)

lemma [*simp*]: $P,C,M,pc \triangleright i \Longrightarrow$ *instrs-of P C M* ! *pc* = *i*

lemma *beforeM*:
$P \vdash C$ *sees M*: *Ts*$\rightarrow T$ = *body in D* $\Longrightarrow$
compP$_2$ *P,D,M,0* $\triangleright$ *compE*$_2$ *body* @ [*Return*]

This lemma executes a single instruction by rewriting:

lemma [*simp*]:
$P,C,M,pc \triangleright$ *instr* $\Longrightarrow$
$(P \vdash (None, h, (vs,ls,C,M,pc) \# frs) -jvm\rightarrow \sigma') =$
$((None, h, (vs,ls,C,M,pc) \# frs) = \sigma' \vee$
$(\exists \sigma.\ exec(P,(None, h, (vs,ls,C,M,pc) \# frs)) = Some\ \sigma \wedge P \vdash \sigma\ -jvm\rightarrow \sigma'))$

5.7.2 Exception tables

definition *pcs* :: *ex-table* $\Rightarrow$ *nat set*
where
pcs xt $\equiv \bigcup (f,t,C,h,d) \in set\ xt.\ \{f\ ..<\ t\}$

lemma *pcs-subset*:
shows $\bigwedge pc\ d.\ pcs(compxE_2\ e\ pc\ d) \subseteq \{pc..<pc+size(compE_2\ e)\}$
and $\bigwedge pc\ d.\ pcs(compxEs_2\ es\ pc\ d) \subseteq \{pc..<pc+size(compEs_2\ es)\}$

lemma [*simp*]: *pcs* [] = {}

lemma [*simp*]: *pcs* (*x*#*xt*) = {*fst x* ..< *fst*(*snd x*)} $\cup$ *pcs xt*

lemma [*simp*]: $pcs(xt_1\ @\ xt_2) = pcs\ xt_1 \cup pcs\ xt_2$

lemma [*simp*]: $pc < pc_0 \vee pc_0+size(compE_2\ e) \leq pc \Longrightarrow pc \notin pcs(compxE_2\ e\ pc_0\ d)$

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-ex-table P C pc* $(xt_0\ @\ xt_1) =$ *match-ex-table P C pc* xt_1

lemma [*simp*]: ⟦ $x \in set\ xt$; $pc \notin pcs\ xt$ ⟧ $\Longrightarrow \neg$ *matches-ex-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-ex-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-ex-entry P C pc' xe*

lemma *match-ex-table-app*[*simp*]:
$\forall xte \in set\ xt_1.\ \neg$ *matches-ex-entry P D pc xte* $\Longrightarrow$

match-ex-table P D pc (xt_1 @ *xt*) = *match-ex-table P D pc xt*

lemma [*simp*]:
∀ *x* ∈ *set xtab*. ¬ *matches-ex-entry P C pc x* ⟹
match-ex-table P C pc xtab = *None*

lemma *match-ex-entry*:
matches-ex-entry P C pc (*start*, *end*, *catch-type*, *handler*) =
(*start* ≤ *pc* ∧ *pc* < *end* ∧ *P* ⊢ *C* ⪯* *catch-type*)

definition *caught* :: *jvm-prog* ⇒ *pc* ⇒ *heap* ⇒ *addr* ⇒ *ex-table* ⇒ *bool* **where**
caught P pc h a xt ⟷
(∃ *entry* ∈ *set xt*. *matches-ex-entry P* (*cname-of h a*) *pc entry*)

definition *beforex* :: *jvm-prog* ⇒ *cname* ⇒ *mname* ⇒ *ex-table* ⇒ *nat set* ⇒ *nat* ⇒ *bool*
((*2*-,/-,/- ▷/ - /'/ -,/-) [*51*,*0*,*0*,*0*,*0*,*51*] *50*) **where**
P,*C*,*M* ▷ *xt* / *I*,*d* ⟷
(∃ xt_0 xt_1. *ex-table-of P C M* = xt_0 @ *xt* @ xt_1 ∧ *pcs* xt_0 ∩ *I* = {} ∧ *pcs xt* ⊆ *I* ∧
(∀ *pc* ∈ *I*. ∀ *C pc*′ *d*′. *match-ex-table P C pc* xt_1 = ⌊(*pc*′,*d*′)⌋ ⟶ *d*′ ≤ *d*))

definition *dummyx* :: *jvm-prog* ⇒ *cname* ⇒ *mname* ⇒ *ex-table* ⇒ *nat set* ⇒ *nat* ⇒ *bool* ((*2*-,-,- ▷/ - /'/-,-) [*51*,*0*,*0*,*0*,*0*,*51*] *50*) **where**
P,*C*,*M* ▷ *xt*/*I*,*d* ⟷ *P*,*C*,*M* ▷ *xt*/*I*,*d*

abbreviation
beforex$_0$ *P C M d I xt* xt_0 xt_1
≡ *ex-table-of P C M* = xt_0 @ *xt* @ xt_1 ∧ *pcs* xt_0 ∩ *I* = {}
∧ *pcs xt* ⊆ *I* ∧ (∀ *pc* ∈ *I*. ∀ *C pc*′ *d*′. *match-ex-table P C pc* xt_1 = ⌊(*pc*′,*d*′)⌋ ⟶ *d*′ ≤ *d*)

lemma *beforex-beforex*$_0$*-eq*:
P,*C*,*M* ▷ *xt* / *I*,*d* ≡ ∃ xt_0 xt_1. *beforex*$_0$ *P C M d I xt* xt_0 xt_1
using *beforex-def* **by** *auto*

lemma *beforexD1*: *P*,*C*,*M* ▷ *xt* / *I*,*d* ⟹ *pcs xt* ⊆ *I*

lemma *beforex-mono*: ⟦ *P*,*C*,*M* ▷ *xt*/*I*,*d*′; *d*′ ≤ *d* ⟧ ⟹ *P*,*C*,*M* ▷ *xt*/*I*,*d*

lemma [*simp*]: *P*,*C*,*M* ▷ *xt*/*I*,*d* ⟹ *P*,*C*,*M* ▷ *xt*/*I*,*Suc d*

lemma *beforex-append*[*simp*]:
pcs xt_1 ∩ *pcs* xt_2 = {} ⟹
P,*C*,*M* ▷ xt_1 @ xt_2/*I*,*d* =
(*P*,*C*,*M* ▷ xt_1/*I*−*pcs* xt_2,*d* ∧ *P*,*C*,*M* ▷ xt_2/*I*−*pcs* xt_1,*d* ∧ *P*,*C*,*M* ▷ xt_1@xt_2/*I*,*d*)

lemma *beforex-appendD1*:
assumes *bx*: *P*,*C*,*M* ▷ xt_1 @ xt_2 @ [(*f*,*t*,*D*,*h*,*d*)] / *I*,*d*
and *pcs*: *pcs* xt_1 ⊆ *J* **and** *JI*: *J* ⊆ *I* **and** *Jpcs*: *J* ∩ *pcs* xt_2 = {}
shows *P*,*C*,*M* ▷ xt_1 / *J*,*d*

lemma *beforex-appendD2*:
assumes *bx*: *P*,*C*,*M* ▷ xt_1 @ xt_2 @ [(*f*,*t*,*D*,*h*,*d*)] / *I*,*d*
and *pcs*: *pcs* xt_2 ⊆ *J* **and** *JI*: *J* ⊆ *I* **and** *Jpcs*: *J* ∩ *pcs* xt_1 = {}
shows *P*,*C*,*M* ▷ xt_2 / *J*,*d*

lemma *beforexM*:
$P \vdash C$ *sees* M: $Ts \to T$ = *body in* $D \Longrightarrow$
$compP_2\ P,D,M \rhd compxE_2\ body\ 0\ 0/\{..<size(compE_2\ body)\},0$

lemma *match-ex-table-SomeD2*:
assumes *met*: *match-ex-table* $P\ D\ pc$ (*ex-table-of* $P\ C\ M$) = $\lfloor(pc',d')\rfloor$
and *bx*: $P,C,M \rhd xt/I,d$
and *nmet*: $\forall x \in set\ xt.\ \neg$ *matches-ex-entry* $P\ D\ pc\ x$ **and** *pcI*: $pc \in I$
shows $d' \leq d$

lemma *match-ex-table-SomeD1*:
$[\![$ *match-ex-table* $P\ D\ pc$ (*ex-table-of* $P\ C\ M$) = $\lfloor(pc',d')\rfloor$;
$P,C,M \rhd xt\ /\ I,d$; $pc \in I$; $pc \notin pcs\ xt\ ]\!] \Longrightarrow d' \leq d$

5.7.3 The correctness proof

definition
handle :: *jvm-prog* ⇒ *cname* ⇒ *mname* ⇒ *addr* ⇒ *heap* ⇒ *val list* ⇒ *val list* ⇒ *nat* ⇒ *frame list*
⇒ *jvm-state* **where**
handle $P\ C\ M\ a\ h\ vs\ ls\ pc\ frs$ = *find-handler* $P\ a\ h$ $((vs,ls,C,M,pc)\ \#\ frs)$

lemma *handle-Cons*:
$[\![\ P,C,M \rhd xt/I,d$; $d \leq size\ vs$; $pc \in I$;
$\forall x \in set\ xt.\ \neg$ *matches-ex-entry* P (*cname-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*:
assumes *bx*: $P,C,M \rhd xt/I,d$ **and** *d*: $d \leq size\ vs$
and *pcI*: $pc \in I$ **and** *pc-not*: $pc \notin pcs\ xt$
shows *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*:
$P_1 \vdash_1 \langle e,(h_0,ls_0)\rangle \Rightarrow \langle ef,(h_1,ls_1)\rangle \Longrightarrow$
$(\bigwedge C\ M\ pc\ v\ xa\ vs\ frs\ I.$
$[\![\ P,C,M,pc \rhd compE_2\ e$; $P,C,M \rhd compxE_2\ e\ pc\ (size\ vs)/I,size\ vs$;
$\{pc..<pc+size(compE_2\ e)\} \subseteq I\ ]\!] \Longrightarrow$
$(ef = Val\ v \longrightarrow$
$P \vdash (None,h_0,(vs,ls_0,C,M,pc)\#frs)\ -jvm\to$
$(None,h_1,(v\#vs,ls_1,C,M,pc+size(compE_2\ e))\#frs))$
$\wedge$
$(ef = Throw\ xa \longrightarrow$
$(\exists pc_1.\ pc \leq pc_1 \wedge pc_1 < pc + size(compE_2\ e) \wedge$
$\neg\ caught\ P\ pc_1\ h_1\ xa\ (compxE_2\ e\ pc\ (size\ vs)) \wedge$
$P \vdash (None,h_0,(vs,ls_0,C,M,pc)\#frs)\ -jvm\to\ handle\ P\ C\ M\ xa\ h_1\ vs\ ls_1\ pc_1\ frs)))$
and $P_1 \vdash_1 \langle es,(h_0,ls_0)\rangle\ [\Rightarrow]\ \langle fs,(h_1,ls_1)\rangle \Longrightarrow$
$(\bigwedge C\ M\ pc\ ws\ xa\ es'\ vs\ frs\ I.$
$[\![\ P,C,M,pc \rhd compEs_2\ es$; $P,C,M \rhd compxEs_2\ es\ pc\ (size\ vs)/I,size\ vs$;
$\{pc..<pc+size(compEs_2\ es)\} \subseteq I\ ]\!] \Longrightarrow$
$(fs = map\ Val\ ws \longrightarrow$

$P \vdash (None,h_0,(vs,ls_0,C,M,pc)\#frs)\ -jvm\rightarrow$
$(None,h_1,(rev\ ws\ @\ vs,ls_1,C,M,pc+size(compEs_2\ es))\#frs))$
$\wedge$
$(fs = map\ Val\ ws\ @\ Throw\ xa\ \#\ es' \longrightarrow$
$(\exists pc_1.\ pc \leq pc_1 \wedge pc_1 < pc + size(compEs_2\ es) \wedge$
$\neg\ caught\ P\ pc_1\ h_1\ xa\ (compxEs_2\ es\ pc\ (size\ vs)) \wedge$
$P \vdash (None,h_0,(vs,ls_0,C,M,pc)\#frs)\ -jvm\rightarrow\ handle\ P\ C\ M\ xa\ h_1\ vs\ ls_1\ pc_1\ frs)))$

lemma *atLeast0AtMost*[*simp*]: $\{0::nat..n\} = \{..n\}$
by *auto*

lemma *atLeast0LessThan*[*simp*]: $\{0::nat..<n\} = \{..<n\}$
by *auto*

fun *exception* :: *'a exp* ⇒ *addr option* **where**
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

5.8 Combining Stages 1 and 2

theory *Compiler*
imports *Correctness1 Correctness2*
begin

definition *J2JVM* :: *J-prog* ⇒ *jvm-prog*
where
J2JVM ≡ $compP_2 \circ compP_1$

theorem *comp-correct*:
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 *size rest* = *max-vars body*
shows $J2JVM\ P \vdash (None,h,[([],vs@rest,C,M,0)])\ -jvm\rightarrow\ (exception\ e',h',[])$

end

5.9 Preservation of Well-Typedness

theory *TypeComp*
imports *Compiler* ../*BV*/*BVSpec*
begin

locale *TC0* =
fixes *P* :: *J$_1$-prog* **and** *mxl* :: *nat*

begin

definition $ty\ E\ e = (THE\ T.\ P,E \vdash_1 e :: T)$

definition $ty_l\ E\ A' = map\ (\lambda i.\ if\ i \in A' \wedge i < size\ E\ then\ OK(E!i)\ else\ Err)\ [0..<mxl]$

definition $ty_i'\ ST\ E\ A = (case\ A\ of\ None \Rightarrow None \mid \lfloor A' \rfloor \Rightarrow Some(ST,\ ty_l\ E\ A'))$

definition $after\ E\ A\ ST\ e = ty_i'\ (ty\ E\ e\ \#\ ST)\ E\ (A \sqcup \mathcal{A}\ e)$

end

lemma (**in** *TC0*) *ty-def2* [*simp*]: $P,E \vdash_1 e :: T \Longrightarrow ty\ E\ e = T$
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*:
$A \subseteq A' \Longrightarrow P \vdash ty_l\ E\ A'\ [\leq_\top]\ ty_l\ E\ A$

lemma (**in** *TC0*) *ty$_i$'-antimono*:
$A \subseteq A' \Longrightarrow P \vdash ty_i'\ ST\ E\ \lfloor A' \rfloor \leq' ty_i'\ ST\ E\ \lfloor A \rfloor$

lemma (**in** *TC0*) *ty$_l$-env-antimono*:
$P \vdash ty_l\ (E@[T])\ A\ [\leq_\top]\ ty_l\ E\ A$

lemma (**in** *TC0*) *ty$_i$'-env-antimono*:
$P \vdash ty_i'\ ST\ (E@[T])\ A \leq' ty_i'\ ST\ E\ A$

lemma (**in** *TC0*) *ty$_i$'-incr*:
$P \vdash ty_i'\ ST\ (E\ @\ [T])\ \lfloor insert\ (size\ E)\ A \rfloor \leq' ty_i'\ ST\ E\ \lfloor A \rfloor$

lemma (**in** *TC0*) *ty$_l$-incr*:
$P \vdash ty_l\ (E\ @\ [T])\ (insert\ (size\ E)\ A)\ [\leq_\top]\ ty_l\ E\ A$

lemma (**in** *TC0*) *ty$_l$-in-types*:
$set\ E \subseteq types\ P \Longrightarrow ty_l\ E\ A \in nlists\ mxl\ (err\ (types\ P))$
locale *TC1* = *TC0*
begin

primrec $compT :: ty\ list \Rightarrow nat\ hyperset \Rightarrow ty\ list \Rightarrow expr_1 \Rightarrow ty_i'\ list$ **and**
$compTs :: ty\ list \Rightarrow nat\ hyperset \Rightarrow ty\ list \Rightarrow expr_1\ list \Rightarrow ty_i'\ list$ **where**
$compT\ E\ A\ ST\ (new\ C) = []$
$\mid compT\ E\ A\ ST\ (Cast\ C\ e) =$
$compT\ E\ A\ ST\ e\ @\ [after\ E\ A\ ST\ e]$
$\mid compT\ E\ A\ ST\ (Val\ v) = []$
$\mid compT\ E\ A\ ST\ (e_1\ «bop»\ e_2) =$
$(let\ ST_1 = ty\ E\ e_1\#ST;\ A_1 = A \sqcup \mathcal{A}\ e_1\ in$
$compT\ E\ A\ ST\ e_1\ @\ [after\ E\ A\ ST\ e_1]\ @$
$compT\ E\ A_1\ ST_1\ e_2\ @\ [after\ E\ A_1\ ST_1\ e_2])$
$\mid compT\ E\ A\ ST\ (Var\ i) = []$

| *compT E A ST (i := e) = compT E A ST e @*
  [after E A ST e, ty$_i$' ST E (A ⊔ 𝒜 e ⊔ ⌊{i}⌋)]
| *compT E A ST (e·F{D}) =*
  compT E A ST e @ [after E A ST e]
| *compT E A ST (e$_1$·F{D} := e$_2$) =*
  (let ST$_1$ = ty E e$_1$#ST; A$_1$ = A ⊔ 𝒜 e$_1$; A$_2$ = A$_1$ ⊔ 𝒜 e$_2$ in
  compT E A ST e$_1$ @ [after E A ST e$_1$] @
  compT E A$_1$ ST$_1$ e$_2$ @ [after E A$_1$ ST$_1$ e$_2$] @
  [ty$_i$' ST E A$_2$])
| *compT E A ST {i:T; e} = compT (E@[T]) (A⊖i) ST e*
| *compT E A ST (e$_1$;;e$_2$) =*
  (let A$_1$ = A ⊔ 𝒜 e$_1$ in
  compT E A ST e$_1$ @ [after E A ST e$_1$, ty$_i$' ST E A$_1$] @
  compT E A$_1$ ST e$_2$)
| *compT E A ST (if (e) e$_1$ else e$_2$) =*
  (let A$_0$ = A ⊔ 𝒜 e; τ = ty$_i$' ST E A$_0$ in
  compT E A ST e @ [after E A ST e, τ] @
  compT E A$_0$ ST e$_1$ @ [after E A$_0$ ST e$_1$, τ] @
  compT E A$_0$ ST e$_2$)
| *compT E A ST (while (e) c) =*
  (let A$_0$ = A ⊔ 𝒜 e; A$_1$ = A$_0$ ⊔ 𝒜 c; τ = ty$_i$' ST E A$_0$ in
  compT E A ST e @ [after E A ST e, τ] @
  compT E A$_0$ ST c @ [after E A$_0$ ST c, ty$_i$' ST E A$_1$, ty$_i$' ST E A$_0$])
| *compT E A ST (throw e) = compT E A ST e @ [after E A ST e]*
| *compT E A ST (e·M(es)) =*
  compT E A ST e @ [after E A ST e] @
  compTs E (A ⊔ 𝒜 e) (ty E e # ST) es
| *compT E A ST (try e$_1$ catch(C i) e$_2$) =*
  compT E A ST e$_1$ @ [after E A ST e$_1$] @
  [ty$_i$' (Class C#ST) E A, ty$_i$' ST (E@[Class C]) (A ⊔ ⌊{i}⌋)] @
  compT (E@[Class C]) (A ⊔ ⌊{i}⌋) ST e$_2$
| *compTs E A ST [] = []*
| *compTs E A ST (e#es) = compT E A ST e @ [after E A ST e] @*
  compTs E (A ⊔ (𝒜 e)) (ty E e # ST) es

definition *compT$_a$ :: ty list ⇒ nat hyperset ⇒ ty list ⇒ expr$_1$ ⇒ ty$_i$' list* **where**
  compT$_a$ E A ST e = compT E A ST e @ [after E A ST e]

end

lemma *compE$_2$-not-Nil*[*simp*]: *compE$_2$ e ≠ []*
lemma (**in** *TC1*) *compT-sizes*[*simp*]:
shows ⋀*E A ST. size*(*compT E A ST e*) = *size*(*compE$_2$ e*) − *1*
and ⋀*E A ST. size*(*compTs E A ST es*) = *size*(*compEs$_2$ es*)

lemma (**in** *TC1*) [*simp*]: ⋀*ST E.* ⌊*τ*⌋ ∉ *set* (*compT E None ST e*)
and [*simp*]: ⋀*ST E.* ⌊*τ*⌋ ∉ *set* (*compTs E None ST es*)

lemma (**in** *TC0*) *pair-eq-ty$_i$'-conv*:
  (⌊(*ST*, *LT*)⌋ = *ty$_i$' ST$_0$ E A*) =
  (*case A of None ⇒ False | Some A ⇒* (*ST = ST$_0$ ∧ LT = ty$_l$ E A*))

lemma (**in** *TC0*) *pair-conv-ty$_i$'*:
  ⌊(*ST*, *ty$_l$ E A*)⌋ = *ty$_i$' ST E* ⌊*A*⌋

lemma (**in** *TC1*) *compT-LT-prefix*:
$\bigwedge E\ A\ ST_0.\ [\![\ \lfloor(ST,LT)\rfloor \in set(compT\ E\ A\ ST_0\ e);\ \mathcal{B}\ e\ (size\ E)\ ]\!]$
$\Longrightarrow P \vdash \lfloor(ST,LT)\rfloor \leq' ty_i'\ ST\ E\ A$
and
$\bigwedge E\ A\ ST_0.\ [\![\ \lfloor(ST,LT)\rfloor \in set(compTs\ E\ A\ ST_0\ es);\ \mathcal{B}s\ es\ (size\ E)\ ]\!]$
$\Longrightarrow P \vdash \lfloor(ST,LT)\rfloor \leq' ty_i'\ ST\ E\ A$

lemma [*iff*]: *OK None* $\in$ *states P mxs mxl*
lemma (**in** *TC0*) *after-in-states*:
assumes *wf*: *wf-prog p P* **and** *wt*: $P,E \vdash_1 e :: T$
and *Etypes*: *set E* $\subseteq$ *types P* **and** *STtypes*: *set ST* $\subseteq$ *types P*
and *stack*: *size ST* + *max-stack e* $\leq$ *mxs*
shows *OK* (*after E A ST e*) $\in$ *states P mxs mxl*

lemma (**in** *TC0*) *OK-*ty_i'*-in-statesI*[*simp*]:
[\![*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*

definition *shift* :: *nat* $\Rightarrow$ *ex-table* $\Rightarrow$ *ex-table*
where
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-compx*E_2:
shows $\bigwedge pc\ pc'\ d.$ *shift pc* (*compx*E_2 *e pc*$'$ *d*) = *compx*E_2 *e* (*pc*$'$ + *pc*) *d*
and $\bigwedge pc\ pc'\ d.$ *shift pc* (*compxEs*$_2$ *es pc*$'$ *d*) = *compxEs*$_2$ *es* (*pc*$'$ + *pc*) *d*

lemma *compx*E_2*-size-convs*[*simp*]:
shows *n* $\neq$ *0* $\Longrightarrow$ *compx*E_2 *e n d* = *shift n* (*compx*E_2 *e 0 d*)
and *n* $\neq$ *0* $\Longrightarrow$ *compxEs*$_2$ *es n d* = *shift n* (*compxEs*$_2$ *es 0 d*)
locale *TC2* = *TC1* +
fixes T_r :: *ty* **and** *mxs* :: *pc*
begin

definition

wt-instrs :: *instr list* ⇒ *ex-table* ⇒ ty_i' *list* ⇒ *bool*
((⊢ -, - /[::]/ -) [*0*,*0*,*51*] *50*) **where**
⊢ *is*,*xt* [::] τs ⟷ *size is* < *size* τs ∧ *pcs xt* ⊆ {*0*..<*size is*} ∧
(∀ *pc*< *size is*. *P*,T_r,*mxs*,*size* τs,*xt* ⊢ *is*!*pc*,*pc* :: τs)

end

notation *TC2*.*wt-instrs* ((-,-,- ⊢/ -, - /[::]/ -) [*50*,*50*,*50*,*50*,*50*,*51*] *50*)

lemma (**in** *TC2*) [*simp*]: $\tau s \neq [] \Longrightarrow \vdash [],[]\ [::]\ \tau s$
lemma [*simp*]: *eff i P pc et None* = []
lemma *wt-instr-appR*:
⟦ *P*,*T*,*m*,*mpc*,*xt* ⊢ *is*!*pc*,*pc* :: τs;
pc < *size is*; *size is* < *size* τs; *mpc* ≤ *size* τs; *mpc* ≤ *mpc'* ⟧
⟹ *P*,*T*,*m*,*mpc'*,*xt* ⊢ *is*!*pc*,*pc* :: $\tau s@\tau s'$

lemma *relevant-entries-shift* [*simp*]:
relevant-entries P i (*pc*+*n*) (*shift n xt*) = *shift n* (*relevant-entries P i pc xt*)

lemma [*simp*]:
xcpt-eff i P (*pc*+*n*) τ (*shift n xt*) =
map (λ(*pc*,τ). (*pc* + *n*, τ)) (*xcpt-eff i P pc* τ *xt*)

lemma [*simp*]:
app_i (*i*, *P*, *pc*, *m*, *T*, τ) ⟹
eff i P (*pc*+*n*) (*shift n xt*) (*Some* τ) =
map (λ(*pc*,τ). (*pc*+*n*,τ)) (*eff i P pc xt* (*Some* τ))

lemma [*simp*]:
xcpt-app i P (*pc*+*n*) *mxs* (*shift n xt*) τ = *xcpt-app i P pc mxs xt* τ

lemma *wt-instr-appL*:
assumes *P*,*T*,*m*,*mpc*,*xt* ⊢ *i*,*pc* :: τs **and** *pc* < *size* τs **and** *mpc* ≤ *size* τs
shows *P*,*T*,*m*,*mpc* + *size* $\tau s'$,*shift* (*size* $\tau s'$) *xt* ⊢ *i*,*pc*+*size* $\tau s'$ :: $\tau s'@\tau s$

lemma *wt-instr-Cons*:
assumes *wti*: *P*,*T*,*m*,*mpc* − *1*,[] ⊢ *i*,*pc* − *1* :: τs
and *pcl*: *0* < *pc* **and** *mpcl*: *0* < *mpc*
and *pcu*: *pc* < *size* τs + *1* **and** *mpcu*: *mpc* ≤ *size* τs + *1*
shows *P*,*T*,*m*,*mpc*,[] ⊢ *i*,*pc* :: $\tau \# \tau s$

lemma *wt-instr-append*:
assumes *wti*: *P*,*T*,*m*,*mpc* − *size* $\tau s'$,[] ⊢ *i*,*pc* − *size* $\tau s'$ :: τs
and *pcl*: *size* $\tau s'$ ≤ *pc* **and** *mpcl*: *size* $\tau s'$ ≤ *mpc*
and *pcu*: *pc* < *size* τs + *size* $\tau s'$ **and** *mpcu*: *mpc* ≤ *size* τs + *size* $\tau s'$
shows *P*,*T*,*m*,*mpc*,[] ⊢ *i*,*pc* :: $\tau s'@\tau s$

lemma *xcpt-app-pcs*:
pc ∉ *pcs xt* ⟹ *xcpt-app i P pc mxs xt* τ

lemma *xcpt-eff-pcs*:
pc ∉ *pcs xt* ⟹ *xcpt-eff i P pc* τ *xt* = []

lemma *pcs-shift*:

$pc < n \Longrightarrow pc \notin pcs\ (shift\ n\ xt)$

lemma *wt-instr-appRx*:
$[\![\ P,T,m,mpc,xt \vdash is!pc,pc :: \tau s;\ pc < size\ is;\ size\ is < size\ \tau s;\ mpc \leq size\ \tau s\ ]\!]$
$\Longrightarrow P,T,m,mpc,xt\ @\ shift\ (size\ is)\ xt' \vdash is!pc,pc :: \tau s$

lemma *wt-instr-appLx*:
$[\![\ P,T,m,mpc,xt \vdash i,pc :: \tau s;\ pc \notin pcs\ xt'\ ]\!]$
$\Longrightarrow P,T,m,mpc,xt'@xt \vdash i,pc :: \tau s$

lemma (**in** *TC2*) *wt-instrs-extR*:
$\vdash is,xt\ [::]\ \tau s \Longrightarrow \vdash is,xt\ [::]\ \tau s\ @\ \tau s'$

lemma (**in** *TC2*) *wt-instrs-ext*:
assumes wt_1: $\vdash is_1,xt_1\ [::]\ \tau s_1@\tau s_2$ **and** wt_2: $\vdash is_2,xt_2\ [::]\ \tau s_2$
and *τs-size*: $size\ \tau s_1 = size\ is_1$
shows $\vdash is_1@is_2,\ xt_1\ @\ shift\ (size\ is_1)\ xt_2\ [::]\ \tau s_1@\tau s_2$
corollary (**in** *TC2*) *wt-instrs-ext2*:
$[\![\ \vdash is_2,xt_2\ [::]\ \tau s_2;\ \vdash is_1,xt_1\ [::]\ \tau s_1@\tau s_2;\ size\ \tau s_1 = size\ is_1\ ]\!]$
$\Longrightarrow \vdash is_1@is_2,\ xt_1\ @\ shift\ (size\ is_1)\ xt_2\ [::]\ \tau s_1@\tau s_2$

corollary (**in** *TC2*) *wt-instrs-ext-prefix* [*trans*]:
$[\![\ \vdash is_1,xt_1\ [::]\ \tau s_1@\tau s_2;\ \vdash is_2,xt_2\ [::]\ \tau s_3;$
$size\ \tau s_1 = size\ is_1;\ prefix\ \tau s_3\ \tau s_2\ ]\!]$
$\Longrightarrow \vdash is_1@is_2,\ xt_1\ @\ shift\ (size\ is_1)\ xt_2\ [::]\ \tau s_1@\tau s_2$

corollary (**in** *TC2*) *wt-instrs-app*:
assumes is_1: $\vdash is_1,xt_1\ [::]\ \tau s_1@[\tau]$
assumes is_2: $\vdash is_2,xt_2\ [::]\ \tau\#\tau s_2$
assumes *s*: $size\ \tau s_1 = size\ is_1$
shows $\vdash is_1@is_2,\ xt_1@shift\ (size\ is_1)\ xt_2\ [::]\ \tau s_1@\tau\#\tau s_2$

corollary (**in** *TC2*) *wt-instrs-app-last*[*trans*]:
assumes $\vdash is_2,xt_2\ [::]\ \tau\#\tau s_2 \vdash is_1,xt_1\ [::]\ \tau s_1$
$last\ \tau s_1 = \tau\ size\ \tau s_1 = size\ is_1{+}1$
shows $\vdash is_1@is_2,\ xt_1@shift\ (size\ is_1)\ xt_2\ [::]\ \tau s_1@\tau s_2$

corollary (**in** *TC2*) *wt-instrs-append-last*[*trans*]:
assumes *wtis*: $\vdash is,xt\ [::]\ \tau s$ **and** *wti*: $P,T_r,mxs,mpc,[] \vdash i,pc :: \tau s$
and *pc*: $pc = size\ is$ **and** *mpc*: $mpc = size\ \tau s$ **and** *is-τs*: $size\ is + 1 < size\ \tau s$
shows $\vdash is@[i],xt\ [::]\ \tau s$

corollary (**in** *TC2*) *wt-instrs-app2*:
$[\![\ \vdash is_2,xt_2\ [::]\ \tau'\#\tau s_2;\ \vdash is_1,xt_1\ [::]\ \tau\#\tau s_1@[\tau'];$
$xt' = xt_1\ @\ shift\ (size\ is_1)\ xt_2;\ size\ \tau s_1{+}1 = size\ is_1\ ]\!]$
$\Longrightarrow \vdash is_1@is_2,xt'\ [::]\ \tau\#\tau s_1@\tau'\#\tau s_2$

corollary (**in** *TC2*) *wt-instrs-app2-simp*[*trans*,*simp*]:
$[\![\ \vdash is_2,xt_2\ [::]\ \tau'\#\tau s_2;\ \vdash is_1,xt_1\ [::]\ \tau\#\tau s_1@[\tau'];\ size\ \tau s_1{+}1 = size\ is_1\ ]\!]$
$\Longrightarrow \vdash is_1@is_2,\ xt_1@shift\ (size\ is_1)\ xt_2\ [::]\ \tau\#\tau s_1@\tau'\#\tau s_2$

corollary (**in** *TC2*) *wt-instrs-Cons*[*simp*]:
$[\![\ \tau s \neq [];\ \vdash [i],[]\ [::]\ [\tau,\tau'];\ \vdash is,xt\ [::]\ \tau'\#\tau s\ ]\!]$
$\Longrightarrow \vdash i\#is,shift\ 1\ xt\ [::]\ \tau\#\tau'\#\tau s$

theory *Jinja*
imports
J/TypeSafe
J/Annotate

J/execute-Bigstep
J/execute-WellType
JVM/JVMDefensive
JVM/JVMListExample
BV/BVExec
BV/LBVJVM
BV/BVNoTypeError
BV/BVExample
Compiler/TypeComp
begin

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.

[2] G. Klein and T. Nipkow. A machine-checked model for a Java-like language, virtual machine and compiler. *ACM Trans. Prog. Lang. Syst.*, 28(4):619–695, 2006.