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

Gerwin Klein Tobias Nipkow

March 12, 2013

Contents

1 Preface **5**
- 1.1 Theory Dependencies . . . 5

2 Jinja Source Language **11**
- 2.1 Auxiliary Definitions . . . 12
- 2.2 Jinja types . . . 14
- 2.3 Class Declarations and Programs . . . 15
- 2.4 Relations between Jinja Types . . . 16
- 2.5 Jinja Values . . . 23
- 2.6 Objects and the Heap . . . 24
- 2.7 Exceptions . . . 27
- 2.8 Expressions . . . 29
- 2.9 Program State . . . 31
- 2.10 Big Step Semantics . . . 32
- 2.11 Small Step Semantics . . . 37
- 2.12 System Classes . . . 42
- 2.13 Generic Well-formedness of programs . . . 43
- 2.14 Weak well-formedness of Jinja programs . . . 46
- 2.15 Equivalence of Big Step and Small Step Semantics . . . 47
- 2.16 Well-typedness of Jinja expressions . . . 53
- 2.17 Runtime Well-typedness . . . 55
- 2.18 Definite assignment . . . 58
- 2.19 Conformance Relations for Type Soundness Proofs . . . 60
- 2.20 Progress of Small Step Semantics . . . 62
- 2.21 Well-formedness Constraints . . . 64
- 2.22 Type Safety Proof . . . 65
- 2.23 Program annotation . . . 68
- 2.24 Example Expressions . . . 69
- 2.25 Code Generation For BigStep . . . 72
- 2.26 Code Generation For WellType . . . 76

3 Jinja Virtual Machine **79**
- 3.1 State of the JVM . . . 80
- 3.2 Instructions of the JVM . . . 81
- 3.3 JVM Instruction Semantics . . . 82
- 3.4 Exception handling in the JVM . . . 85
- 3.5 Program Execution in the JVM . . . 86

3.6 A Defensive JVM . . . 88
3.7 Example for generating executable code from JVM semantics . . . 92

4 Bytecode Verifier **97**
4.1 Semilattices . . . 98
4.2 The Error Type . . . 102
4.3 More about Options . . . 105
4.4 Products as Semilattices . . . 106
4.5 Fixed Length Lists . . . 107
4.6 Typing and Dataflow Analysis Framework . . . 111
4.7 More on Semilattices . . . 112
4.8 Lifting the Typing Framework to err, app, and eff . . . 114
4.9 Kildall's Algorithm . . . 116
4.10 The Lightweight Bytecode Verifier . . . 119
4.11 Correctness of the LBV . . . 123
4.12 Completeness of the LBV . . . 124
4.13 The Jinja Type System as a Semilattice . . . 126
4.14 The JVM Type System as Semilattice . . . 128
4.15 Effect of Instructions on the State Type . . . 131
4.16 Monotonicity of eff and app . . . 138
4.17 The Bytecode Verifier . . . 139
4.18 The Typing Framework for the JVM . . . 141
4.19 Kildall for the JVM . . . 143
4.20 LBV for the JVM . . . 144
4.21 BV Type Safety Invariant . . . 146
4.22 BV Type Safety Proof . . . 149
4.23 Welltyped Programs produce no Type Errors . . . 155
4.24 Example Welltypings . . . 158

5 Compilation **165**
5.1 An Intermediate Language . . . 166
5.2 Well-Formedness of Intermediate Language . . . 170
5.3 Program Compilation . . . 173
5.4 Compilation Stage 1 . . . 179
5.5 Correctness of Stage 1 . . . 180
5.6 Compilation Stage 2 . . . 182
5.7 Correctness of Stage 2 . . . 185
5.8 Combining Stages 1 and 2 . . . 189
5.9 Preservation of Well-Typedness . . . 190

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.

A tabled implementation of the reflexive transitive closure **theory** *Transitive-Closure-Table*

```
imports Main
begin

inductive rtrancl-path :: ('a ⇒ 'a ⇒ bool) ⇒ 'a ⇒ 'a list ⇒ 'a ⇒ bool
  for r :: 'a ⇒ 'a ⇒ bool
where
  base: rtrancl-path r x [] x
| step: r x y ⟹ rtrancl-path r y ys z ⟹ rtrancl-path r x (y # ys) z

lemma rtranclp-eq-rtrancl-path: r^** x y = (∃xs. rtrancl-path r x xs y)
proof
  assume r^** x y
  then show ∃xs. rtrancl-path r x xs y
  proof (induct rule: converse-rtranclp-induct)
    case base
    have rtrancl-path r y [] y by (rule rtrancl-path.base)
    then show ?case ..
  next
    case (step x z)
    from ‹∃xs. rtrancl-path r z xs y›
    obtain xs where rtrancl-path r z xs y ..
    with ‹r x z› have rtrancl-path r x (z # xs) y
      by (rule rtrancl-path.step)
    then show ?case ..
  qed
next
  assume ∃xs. rtrancl-path r x xs y
  then obtain xs where rtrancl-path r x xs y ..
  then show r^** x y
  proof induct
    case (base x)
    show ?case by (rule rtranclp.rtrancl-refl)
  next
    case (step x y ys z)
    from ‹r x y› ‹r^** y z› show ?case
      by (rule converse-rtranclp-into-rtranclp)
  qed
qed

lemma rtrancl-path-trans:
  assumes xy: rtrancl-path r x xs y
  and yz: rtrancl-path r y ys z
  shows rtrancl-path r x (xs @ ys) z using xy yz
proof (induct arbitrary: z)
  case (base x)
  then show ?case by simp
next
  case (step x y xs)
  then have rtrancl-path r y (xs @ ys) z
    by simp
  with ‹r x y› have rtrancl-path r x (y # (xs @ ys)) z
    by (rule rtrancl-path.step)
  then show ?case by simp
```

```
qed

lemma rtrancl-path-appendE:
  assumes xz: rtrancl-path r x (xs @ y # ys) z
  obtains rtrancl-path r x (xs @ [y]) y and rtrancl-path r y ys z using xz
proof (induct xs arbitrary: x)
  case Nil
  then have rtrancl-path r x (y # ys) z by simp
  then obtain xy: r x y and yz: rtrancl-path r y ys z
    by cases auto
  from xy have rtrancl-path r x [y] y
    by (rule rtrancl-path.step [OF - rtrancl-path.base])
  then have rtrancl-path r x ([] @ [y]) y by simp
  then show ?thesis using yz by (rule Nil)
next
  case (Cons a as)
  then have rtrancl-path r x (a # (as @ y # ys)) z by simp
  then obtain xa: r x a and az: rtrancl-path r a (as @ y # ys) z
    by cases auto
  show ?thesis
  proof (rule Cons(1) [OF - az])
    assume rtrancl-path r y ys z
    assume rtrancl-path r a (as @ [y]) y
    with xa have rtrancl-path r x (a # (as @ [y])) y
      by (rule rtrancl-path.step)
    then have rtrancl-path r x ((a # as) @ [y]) y
      by simp
    then show ?thesis using ‹rtrancl-path r y ys z›
      by (rule Cons(2))
  qed
qed

lemma rtrancl-path-distinct:
  assumes xy: rtrancl-path r x xs y
  obtains xs' where rtrancl-path r x xs' y and distinct (x # xs') using xy
proof (induct xs rule: measure-induct-rule [of length])
  case (less xs)
  show ?case
  proof (cases distinct (x # xs))
    case True
    with ‹rtrancl-path r x xs y› show ?thesis by (rule less)
  next
    case False
    then have ∃ as bs cs a. x # xs = as @ [a] @ bs @ [a] @ cs
      by (rule not-distinct-decomp)
    then obtain as bs cs a where xxs: x # xs = as @ [a] @ bs @ [a] @ cs
      by iprover
    show ?thesis
    proof (cases as)
      case Nil
      with xxs have x: x = a and xs: xs = bs @ a # cs
        by auto
      from x xs ‹rtrancl-path r x xs y› have cs: rtrancl-path r x cs y
        by (auto elim: rtrancl-path-appendE)
```

```
      from xs have length cs < length xs by simp
      then show ?thesis
        by (rule less(1)) (iprover intro: cs less(2))+
    next
      case (Cons d ds)
      with xxs have xs: xs = ds @ a # (bs @ [a] @ cs)
        by auto
      with ‹rtrancl-path r x xs y› obtain xa: rtrancl-path r x (ds @ [a]) a
        and ay: rtrancl-path r a (bs @ a # cs) y
        by (auto elim: rtrancl-path-appendE)
      from ay have rtrancl-path r a cs y by (auto elim: rtrancl-path-appendE)
      with xa have xy: rtrancl-path r x ((ds @ [a]) @ cs) y
        by (rule rtrancl-path-trans)
      from xs have length ((ds @ [a]) @ cs) < length xs by simp
      then show ?thesis
        by (rule less(1)) (iprover intro: xy less(2))+
    qed
  qed
qed

inductive rtrancl-tab :: ('a ⇒ 'a ⇒ bool) ⇒ 'a list ⇒ 'a ⇒ 'a ⇒ bool
  for r :: 'a ⇒ 'a ⇒ bool
where
  base: rtrancl-tab r xs x x
| step: x ∉ set xs ⟹ r x y ⟹ rtrancl-tab r (x # xs) y z ⟹ rtrancl-tab r xs x z

lemma rtrancl-path-imp-rtrancl-tab:
  assumes path: rtrancl-path r x xs y
  and x: distinct (x # xs)
  and ys: ({x} ∪ set xs) ∩ set ys = {}
  shows rtrancl-tab r ys x y using path x ys
proof (induct arbitrary: ys)
  case base
  show ?case by (rule rtrancl-tab.base)
next
  case (step x y zs z)
  then have x ∉ set ys by auto
  from step have distinct (y # zs) by simp
  moreover from step have ({y} ∪ set zs) ∩ set (x # ys) = {}
    by auto
  ultimately have rtrancl-tab r (x # ys) y z
    by (rule step)
  with ‹x ∉ set ys› ‹r x y›
  show ?case by (rule rtrancl-tab.step)
qed

lemma rtrancl-tab-imp-rtrancl-path:
  assumes tab: rtrancl-tab r ys x y
  obtains xs where rtrancl-path r x xs y using tab
proof induct
  case base
  from rtrancl-path.base show ?case by (rule base)
next
  case step show ?case by (iprover intro: step rtrancl-path.step)
```

qed

lemma *rtranclp-eq-rtrancl-tab-nil*: r^{**} *x y* = *rtrancl-tab r* [] *x y*
proof
  assume r^{**} *x y*
  then obtain *xs* **where** *rtrancl-path r x xs y*
    by (*auto simp add*: *rtranclp-eq-rtrancl-path*)
  then obtain *xs'* **where** *xs'*: *rtrancl-path r x xs' y*
    and *distinct*: *distinct* (*x* # *xs'*)
    by (*rule rtrancl-path-distinct*)
  have ({*x*} ∪ *set xs'*) ∩ *set* [] = {} **by** *simp*
  with *xs' distinct* **show** *rtrancl-tab r* [] *x y*
    by (*rule rtrancl-path-imp-rtrancl-tab*)
next
  assume *rtrancl-tab r* [] *x y*
  then obtain *xs* **where** *rtrancl-path r x xs y*
    by (*rule rtrancl-tab-imp-rtrancl-path*)
  then show r^{**} *x y*
    by (*auto simp add*: *rtranclp-eq-rtrancl-path*)
qed

declare *rtranclp-rtrancl-eq*[*code del*]
declare *rtranclp-eq-rtrancl-tab-nil*[*THEN iffD2*, *code-pred-intro*]

code-pred *rtranclp* **using** *rtranclp-eq-rtrancl-tab-nil* [*THEN iffD1*] **by** *fastforce*

1.1.1 A simple example

datatype *ty* = *A* | *B* | *C*

inductive *test* :: *ty* ⇒ *ty* ⇒ *bool*
where
  test A B
| *test B A*
| *test B C*

Invoking with the predicate compiler and the generic code generator

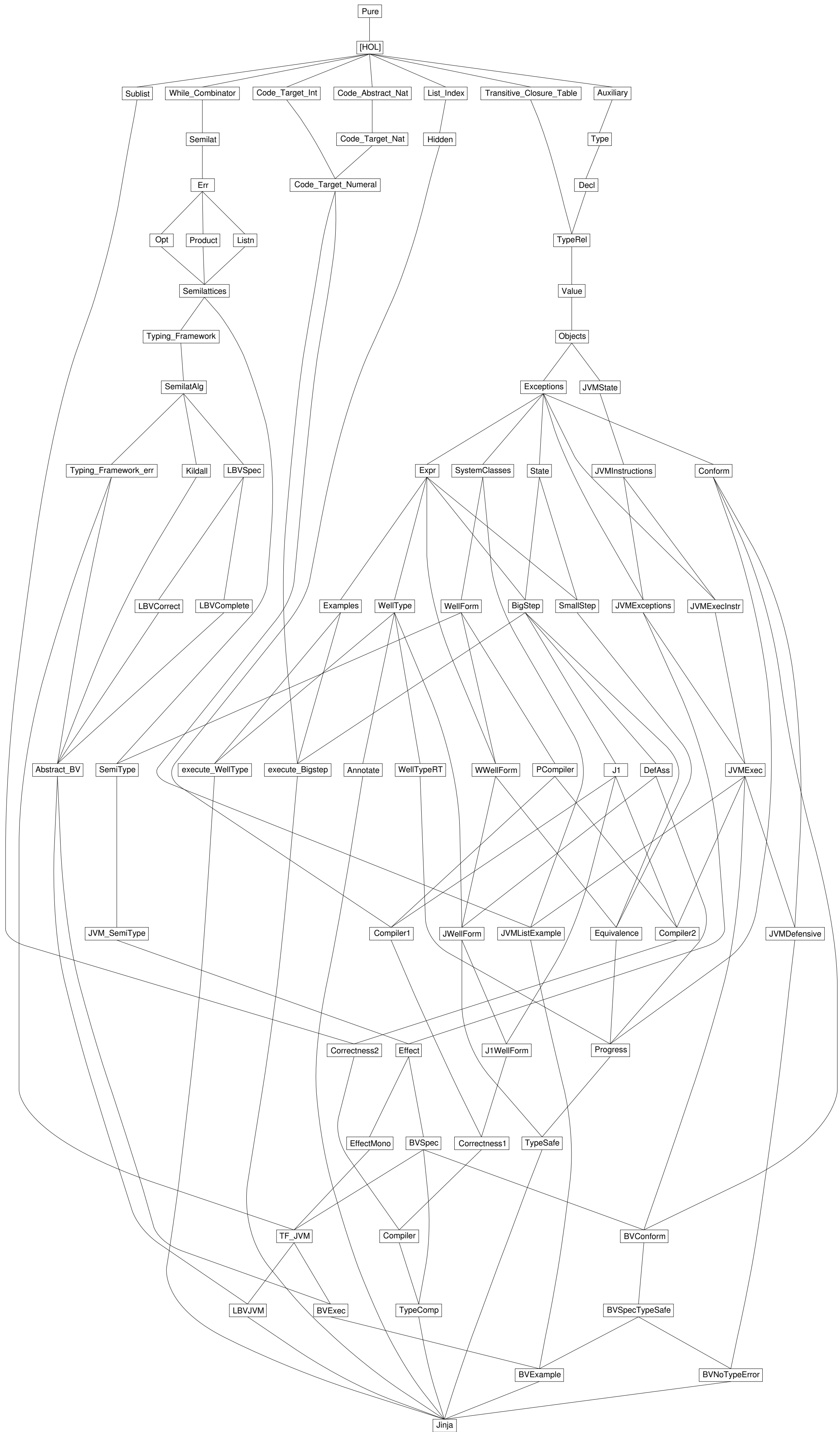
code-pred *test* **.**

values {*x*. $test^{**}$ *A C*}
values {*x*. $test^{**}$ *C A*}
values {*x*. $test^{**}$ *A x*}
values {*x*. $test^{**}$ *x C*}

value $test^{**}$ *A C*
value $test^{**}$ *C A*

hide-type *ty*
hide-const *test A B C*

end


Pure
[HOL]
Sublist
While_Combinator
Code_Target_Int
Code_Abstract_Nat
List_Index
Transitive_Closure_Table
Auxiliary
Semilat
Code_Target_Nat
Hidden
Type
Err
Code_Target_Numeral
Decl
Opt
Product
Listn
TypeRel
Semilattices
Value
Typing_Framework
Objects
SemilatAlg
Exceptions
JVMState
Typing_Framework_err
Kildall
LBVSpec
Expr
SystemClasses
State
JVMInstructions
Conform
LBVCorrect
LBVComplete
Examples
WellType
WellForm
BigStep
SmallStep
JVMExceptions
JVMExecInstr
Abstract_BV
SemiType
execute_WellType
execute_Bigstep
Annotate
WellTypeRT
WWellForm
PCompiler
J1
DefAss
JVMExec
JVM_SemiType
Compiler1
JWellForm
JVMListExample
Equivalence
Compiler2
JVMDefensive
Correctness2
Effect
J1WellForm
Progress
EffectMono
BVSpec
Correctness1
TypeSafe
TF_JVM
Compiler
BVConform
LBVJVM
BVExec
TypeComp
BVSpecTypeSafe
BVExample
BVNoTypeError
Jinja

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* ((⌊-⌋))

2.1.1 *distinct-fst*

definition *distinct-fst* :: $('a \times 'b)\ list \Rightarrow bool$
where
distinct-fst ≡ *distinct* ∘ *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*:
$list\text{-}all2\ (fun\text{-}of\ S)\ (x\#xs)\ ys =$
$(\exists z\ zs.\ ys = z\#zs \wedge (x,z) \in S \wedge list\text{-}all2\ (fun\text{-}of\ S)\ xs\ zs)$

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

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

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

lemma *rel-list-all2-trans*:

$\llbracket \bigwedge a\ b\ c.\ \llbracket(a,b) \in R;\ (b,c) \in S\rrbracket \Longrightarrow (a,c) \in T;$
list-all2 (*fun-of* R) *as* *bs*; *list-all2* (*fun-of* S) *bs* *cs*$\rrbracket$
$\Longrightarrow$ *list-all2* (*fun-of* T) *as* *cs*

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

lemma *rel-list-all2-nthD*:
$\llbracket$ *list-all2* (*fun-of* S) *xs* *ys*; p < *size* *xs* $\rrbracket \Longrightarrow$ (*xs*!p,*ys*!p) $\in S$

lemma *rel-list-all2I*:
$\llbracket$ *length* a = *length* b; $\bigwedge n.\ n$ < *length* $a \Longrightarrow (a!n,b!n) \in S$ $\rrbracket \Longrightarrow$ *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 ≡ $T = NT \vee (\exists C.\ T = Class\ C)$

lemma [*iff*]: *is-refT NT*
lemma [*iff*]: *is-refT*(*Class C*)
lemma *refTE*:
$[\![$*is-refT T*; $T = NT \Longrightarrow P$; $\bigwedge C.\ T = Class\ C \Longrightarrow P\ ]\!] \Longrightarrow P$
lemma *not-refTE*:
$[\![\ \neg$*is-refT T*; $T = Void \vee T = Boolean \vee T = Integer \Longrightarrow P\ ]\!] \Longrightarrow P$
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**
~~/src/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* (- ⊢ - $\prec^1$ - [71,71,71] 70)
for *P* :: *'m prog*
where
$P \vdash C \prec^1 D \equiv (C,D) \in subcls1\ P$
| *subcls1I*: ⟦*class P C* = *Some* (*D*,*rest*); *C* ≠ *Object*⟧ ⟹ $P \vdash C \prec^1 D$

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

lemma *subcls1D*: $P \vdash C \prec^1 D \Longrightarrow C \neq Object \wedge (\exists fs\ ms.\ class\ P\ C = Some\ (D,fs,ms))$
lemma [*iff*]: $\neg\ P \vdash Object \prec^1 C$
lemma [*iff*]: $(P \vdash Object \preceq^* C) = (C = Object)$
lemma *subcls1-def2*:
subcls1 P =
(*SIGMA C*:{*C*. *is-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*]: $P \vdash T \leq T$
| *widen-subcls*: $P \vdash C \preceq^* D \Longrightarrow P \vdash Class\ C \leq Class\ D$
| *widen-null*[*iff*]: $P \vdash NT \leq Class\ C$

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

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

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

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

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

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

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

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

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

definition *Method* :: $'m\ prog \Rightarrow cname \Rightarrow mname \Rightarrow ty\ list \Rightarrow ty \Rightarrow 'm \Rightarrow cname \Rightarrow bool$
   $(\text{-} \vdash \text{-}\ sees\ \text{-}:\ \text{-}\rightarrow\text{-}\ =\ \text{-}\ in\ \text{-}\ [51,51,51,51,51,51,51]\ 50)$
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* :: $'m\ prog \Rightarrow cname \Rightarrow mname \Rightarrow bool\ (\text{-} \vdash \text{-}\ has\ \text{-}\ [51,0,51]\ 50)$
where
  $P \vdash C\ has\ M \equiv \exists Ts\ T\ m\ D.\ P \vdash C\ sees\ M{:}Ts \rightarrow T = m\ in\ D$

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

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

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

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

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

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

2.4.4 Field lookup

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

lemma *has-fields-fun*:
assumes *1*: $P \vdash C\ has\text{-}fields\ FDTs$
shows $\bigwedge FDTs'.\ P \vdash C\ has\text{-}fields\ FDTs' \Longrightarrow FDTs' = FDTs$

lemma *all-fields-in-has-fields*:
assumes *sub*: $P \vdash C\ has\text{-}fields\ FDTs$
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*: $P \vdash C\ has\text{-}fields\ FDTs$
shows $((F,D),T) \in set\ FDTs \Longrightarrow P \vdash C \preceq^* D$

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

lemma *has-fields-mono-lem*:

assumes *sub*: $P \vdash D \preceq^* C$
shows $P \vdash C$ *has-fields FDTs*
$\Longrightarrow \exists pre.\ P \vdash D$ *has-fields pre@FDTs* $\wedge$ *dom(map-of pre)* $\cap$ *dom(map-of FDTs)* $= \{\}$

definition *has-field* :: *'m prog* $\Rightarrow$ *cname* $\Rightarrow$ *vname* $\Rightarrow$ *ty* $\Rightarrow$ *cname* $\Rightarrow$ *bool*
(- $\vdash$ - *has* -:- *in* - [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-of FDTs* $(F,D) =$ *Some* T

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

definition *sees-field* :: *'m prog* $\Rightarrow$ *cname* $\Rightarrow$ *vname* $\Rightarrow$ *ty* $\Rightarrow$ *cname* $\Rightarrow$ *bool*
(- $\vdash$ - *sees* -:- *in* - [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-of (*map* $(\lambda((F,D),T).\ (F,(D,T)))$ *FDTs*) $F =$ *Some*(D,T)

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

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*:
$P \vdash C$ *sees* $F{:}T$ *in* $D \Longrightarrow P \vdash D$ *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* $M{:}Ts \rightarrow T = m$ *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: $Ts \rightarrow T = m$ *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*) ^**
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*) ^* = {(*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*]:
  $\llbracket$ $P \vdash C$ *has-fields FDTs*; *map-of FDTs* (*F*, *D*) = $\lfloor T \rfloor$ $\rrbracket$
  $\Longrightarrow P \vdash C$ *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*]:
  $\llbracket$ $P \vdash C$ *has-fields FDTs*; *map-of* (*map* (λ((*F*, *D*), *T*). (*F*, *D*, *T*)) *FDTs*) *F* = $\lfloor (D, T) \rfloor$ $\rrbracket$
  $\Longrightarrow P \vdash C$ *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-def 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*) = (λ*T*. $P \vdash C$ *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-def 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 ≡ *map-of* ∘ *map* (λ(*F*,*T*). (*F*,*default-val T*))

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

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

2.6.2 Heap

type-synonym *heap* = *addr* ⇀ *obj*

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

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

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

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

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

lemma *new-Addr-SomeD*:

$new\text{-}Addr\ h = Some\ a \Longrightarrow h\ a = None$

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

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

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

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**
$typeof\ v == typeof\text{-}h\ empty\ v$

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

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

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 $gen\text{-}new\text{-}Addr :: heap \Rightarrow addr \Rightarrow addr\ option$
where $gen\text{-}new\text{-}Addr\ h\ n \equiv if\ \exists\, a.\ a \geq n \wedge h\ a = None\ then\ Some(LEAST\ a.\ a \geq n \wedge h\ a = None)$
$else\ None$

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

lemma *gen-new-Addr-code* [*code*]:
$gen\text{-}new\text{-}Addr\ h\ n = (if\ h\ n = None\ then\ Some\ n\ else\ gen\text{-}new\text{-}Addr\ h\ (Suc\ n))$
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 ≡ *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*; $C \in$ *sys-xcpts* $]\!] \Longrightarrow$ *cname-of h* (*addr-of-sys-xcpt C*) $= C$

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

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

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

lemma *preallocated-hext*:
$[\![$ *preallocated h*; $h \trianglelefteq h'$ $]\!] \Longrightarrow$ *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*]:
$(binop(Add,v_1,v_2) = Some\ v) = (\exists\, i_1\ i_2.\ v_1 = Intg\ i_1 \wedge v_2 = Intg\ i_2 \wedge v = Intg(i_1+i_2))$

2.8.1 Syntactic sugar

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*) = {}
| $fv(e_1 \ll bop \gg e_2) = fv\ e_1 \cup fv\ e_2$
| *fv*(*Var V*) = {*V*}
| *fv*(*LAss V e*) = {*V*} ∪ *fv e*
| *fv*(*e*•*F*{*D*}) = *fv e*
| $fv(e_1{\bullet}F\{D\}{:=}e_2) = fv\ e_1 \cup fv\ e_2$
| *fv*(*e*•*M*(*es*)) = *fv e* ∪ *fvs es*
| *fv*({*V*:*T*; *e*}) = *fv e* − {*V*}
| $fv(e_1;;e_2) = fv\ e_1 \cup fv\ e_2$
| $fv(if\ (b)\ e_1\ else\ e_2) = fv\ b \cup fv\ e_1 \cup fv\ e_2$
| *fv*(*while* (*b*) *e*) = *fv b* ∪ *fv e*
| *fv*(*throw e*) = *fv e*
| $fv(try\ e_1\ catch(C\ V)\ e_2) = fv\ e_1 \cup (fv\ e_2 - \{V\})$
| *fvs*([]) = {}
| *fvs*(*e*#*es*) = *fv e* ∪ *fvs es*

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

2.9 Program State

theory *State* **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* $\preceq^*$ *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* $\preceq^*$ *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 ⊢ ⟨*Val v*,*s*⟩ ⇒ ⟨*Val v*,*s*⟩

| *BinOp*:
⟦ *P* ⊢ ⟨e_1,s_0⟩ ⇒ ⟨*Val* v_1,s_1⟩; *P* ⊢ ⟨e_2,s_1⟩ ⇒ ⟨*Val* v_2,s_2⟩; *binop*(*bop*,v_1,v_2) = *Some v* ⟧
⟹ *P* ⊢ ⟨e_1 «*bop*» e_2,s_0⟩⇒⟨*Val v*,s_2⟩

| *BinOpThrow1*:
P ⊢ ⟨e_1,s_0⟩ ⇒ ⟨*throw e*,s_1⟩ ⟹
P ⊢ ⟨e_1 «*bop*» e_2, s_0⟩ ⇒ ⟨*throw e*,s_1⟩

| *BinOpThrow2*:
⟦ *P* ⊢ ⟨e_1,s_0⟩ ⇒ ⟨*Val* v_1,s_1⟩; *P* ⊢ ⟨e_2,s_1⟩ ⇒ ⟨*throw e*,s_2⟩ ⟧
⟹ *P* ⊢ ⟨e_1 «*bop*» e_2,s_0⟩ ⇒ ⟨*throw e*,s_2⟩

| *Var*:
l V = *Some v* ⟹
P ⊢ ⟨*Var V*,(*h*,*l*)⟩ ⇒ ⟨*Val v*,(*h*,*l*)⟩

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

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

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

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

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

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

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

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

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

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

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

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

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

$P \vdash \langle body,(h_2,l_2')\rangle \Rightarrow \langle e',(h_3,l_3)\rangle$ ⟧
$\Longrightarrow P \vdash \langle e{\cdot}M(ps),s_0\rangle \Rightarrow \langle e',(h_3,l_2)\rangle$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

2.10.1 Final expressions

definition *final* :: $'a\ exp \Rightarrow bool$
where
$final\ e\ \equiv\ (\exists v.\ e = Val\ v) \vee (\exists a.\ e = Throw\ a)$

definition *finals*:: $'a\ exp\ list \Rightarrow bool$
where
$finals\ es\ \equiv\ (\exists vs.\ es = map\ Val\ vs) \vee (\exists vs\ a\ es'.\ es = map\ Val\ vs\ @\ Throw\ a\ \#\ es')$

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

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

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

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

lemma *eval-finalId*: *final* $e \Longrightarrow P \vdash \langle e,s \rangle \Rightarrow \langle e,s \rangle$

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

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

end

2.11 Small Step Semantics

```
theory SmallStep
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 ⊢ ⟨e,s⟩ → ⟨e′,s′⟩ ≡ ((e,s), e′,s′) ∈ red P
| P ⊢ ⟨es,s⟩ [→] ⟨es′,s′⟩ ≡ ((es,s), es′,s′) ∈ reds P

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

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

| CastRed:
  P ⊢ ⟨e,s⟩ → ⟨e′,s′⟩ =⇒
  P ⊢ ⟨Cast C e, s⟩ → ⟨Cast C e′, s′⟩

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

| RedCast:
 ⟦ hp s a = Some(D,fs); P ⊢ D ⪯∗ C ⟧
  =⇒ P ⊢ ⟨Cast C (addr a), s⟩ → ⟨addr a, s⟩

| RedCastFail:
  ⟦ hp s a = Some(D,fs); ¬ P ⊢ D ⪯∗ C ⟧
```

$\Longrightarrow P \vdash \langle Cast\ C\ (addr\ a),\ s\rangle \rightarrow \langle THROW\ ClassCast,\ s\rangle$

| *BinOpRed1*:
$P \vdash \langle e,s\rangle \rightarrow \langle e',s'\rangle \Longrightarrow$
$P \vdash \langle e \ll bop \gg e_2,\ s\rangle \rightarrow \langle e' \ll bop \gg e_2,\ s'\rangle$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

— Exception propagation

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

2.11.1 The reflexive transitive closure

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

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

where $P \vdash \langle es,s\rangle\ [\rightarrow]{*}\ \langle es',s'\rangle \equiv ((es,s),\ es',s') \in (reds\ P)^*$

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

2.11.2 Some easy lemmas

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

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

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

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

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

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 ≡ λ(C,(D,fs,ms)).
(∀ f∈set fs. wf-fdecl P f) ∧ distinct-fst fs ∧
(∀ m∈set ms. wf-mdecl wf-md P C m) ∧ distinct-fst ms ∧
(C ≠ Object ⟶
is-class P D ∧ ¬ P ⊢ D ⪯ C ∧*
(∀ (M,Ts,T,m)∈set ms.
∀ D' Ts' T' m'. P ⊢ D sees M:Ts' → T' = m' in D' ⟶
P ⊢ Ts' [≤] Ts ∧ P ⊢ T ≤ T'))

definition *wf-syscls* :: *'m prog ⇒ bool*
where
wf-syscls P ≡ {Object} ∪ sys-xcpts ⊆ set(map fst P)

definition *wf-prog* :: *'m wf-mdecl-test ⇒ 'm prog ⇒ bool*
where
wf-prog wf-md P ≡ wf-syscls P ∧ (∀ c ∈ set P. wf-cdecl wf-md P c) ∧ distinct-fst P

2.13.1 Well-formedness lemmas

lemma *class-wf*:
⟦class P C = Some c; wf-prog wf-md P⟧ ⟹ wf-cdecl wf-md P (C,c)

lemma *class-Object* [*simp*]:
wf-prog wf-md P ⟹ ∃ C fs ms. class P Object = Some (C,fs,ms)

lemma *is-class-Object* [*simp*]:
wf-prog wf-md P ⟹ is-class P Object

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

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

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

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

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

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

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

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

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

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

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

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

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

2.13.2 Well-formedness and method lookup

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

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

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

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

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

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

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

2.13.3 Well-formedness and field lookup

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

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

lemma *sees-field-is-type*:
$[\![P \vdash C \textit{ sees } F{:}T \textit{ in } D;\ \textit{wf-prog wf-md } P]\!] \Longrightarrow \textit{is-type } P\ T$
lemma *wf-syscls*:
$\textit{set SystemClasses} \subseteq \textit{set } P \Longrightarrow \textit{wf-syscls } P$
end

2.14 Weak well-formedness of Jinja programs

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

definition *wwf-J-mdecl* :: *J-prog* ⇒ *cname* ⇒ *J-mb mdecl* ⇒ *bool*
where
wwf-J-mdecl P C ≡ λ(*M*,*Ts*,*T*,(*pns*,*body*)).
length Ts = *length pns* ∧ *distinct pns* ∧ *this* ∉ *set pns* ∧ *fv body* ⊆ {*this*} ∪ *set pns*

lemma *wwf-J-mdecl*[*simp*]:
wwf-J-mdecl P C (*M*,*Ts*,*T*,*pns*,*body*) =
(*length Ts* = *length pns* ∧ *distinct pns* ∧ *this* ∉ *set pns* ∧ *fv body* ⊆ {*this*} ∪ *set pns*)
abbreviation
wwf-J-prog :: *J-prog* ⇒ *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*:
$P \vdash \langle e,s\rangle \to* \langle e',s'\rangle \Longrightarrow P \vdash \langle\ e \ll bop\gg e_2,\ s\rangle \to* \langle e' \ll bop\gg e_2,\ s'\rangle$
lemma *BinOp2Reds*:
$P \vdash \langle e,s\rangle \to* \langle e',s'\rangle \Longrightarrow P \vdash \langle (Val\ v) \ll bop\gg e,\ s\rangle \to* \langle (Val\ v) \ll bop\gg e',\ s'\rangle$
lemma *BinOpRedsVal*:
$[\![\ P \vdash \langle e_1,s_0\rangle \to* \langle Val\ v_1,s_1\rangle;\ P \vdash \langle e_2,s_1\rangle \to* \langle Val\ v_2,s_2\rangle;\ binop(bop,v_1,v_2) = Some\ v\ ]\!]$
$\Longrightarrow P \vdash \langle e_1 \ll bop\gg e_2,\ s_0\rangle \to* \langle Val\ v,s_2\rangle$
lemma *BinOpRedsThrow1*:
$P \vdash \langle e,s\rangle \to* \langle throw\ e',s'\rangle \Longrightarrow P \vdash \langle e \ll bop\gg e_2,\ s\rangle \to* \langle throw\ e',\ s'\rangle$
lemma *BinOpRedsThrow2*:
$[\![\ P \vdash \langle e_1,s_0\rangle \to* \langle Val\ v_1,s_1\rangle;\ P \vdash \langle e_2,s_1\rangle \to* \langle throw\ e,s_2\rangle]\!]$
$\Longrightarrow P \vdash \langle e_1 \ll bop\gg e_2,\ s_0\rangle \to* \langle throw\ e,s_2\rangle$

FAcc

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

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

FAss

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

;;

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

If

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

While

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

Throw

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

InitBlock

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

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

try-catch

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

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

Call

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

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

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

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

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

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

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

An now the actual method call reduction lemmas.

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

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

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

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

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

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

The main Theorem

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

2.15.2 Big steps simulates small step

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

The big step equivalent of *RedWhile*:

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

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

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

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

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

2.15.3 Equivalence

And now, the crowning achievement:

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

end

2.16 Well-typedness of Jinja expressions

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

type-synonym
env = *vname* ⇀ *ty*

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

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

| *WTCast*:
⟦ *P*,*E* ⊢ *e* :: *Class D*; *is-class P C*; *P* ⊢ *C* ⪯* *D* ∨ *P* ⊢ *D* ⪯* *C* ⟧
⟹ *P*,*E* ⊢ *Cast C e* :: *Class C*

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

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

| *WTBinOpEq*:
⟦ *P*,*E* ⊢ e_1 :: T_1; *P*,*E* ⊢ e_2 :: T_2; *P* ⊢ T_1 ≤ T_2 ∨ *P* ⊢ T_2 ≤ T_1 ⟧
⟹ *P*,*E* ⊢ e_1 «*Eq*» e_2 :: *Boolean*

| *WTBinOpAdd*:
⟦ *P*,*E* ⊢ e_1 :: *Integer*; *P*,*E* ⊢ e_2 :: *Integer* ⟧
⟹ *P*,*E* ⊢ e_1 «*Add*» e_2 :: *Integer*

| *WTLAss*:
⟦ *E V* = *Some T*; *P*,*E* ⊢ *e* :: *T'*; *P* ⊢ *T'* ≤ *T*; *V* ≠ *this* ⟧
⟹ *P*,*E* ⊢ *V*:=*e* :: *Void*

| *WTFAcc*:
⟦ *P*,*E* ⊢ *e* :: *Class C*; *P* ⊢ *C sees F*:*T in D* ⟧
⟹ *P*,*E* ⊢ *e*•*F*{*D*} :: *T*

| *WTFAss*:
⟦ *P*,*E* ⊢ e_1 :: *Class C*; *P* ⊢ *C sees F*:*T in D*; *P*,*E* ⊢ e_2 :: *T'*; *P* ⊢ *T'* ≤ *T* ⟧
⟹ *P*,*E* ⊢ e_1•*F*{*D*}:=e_2 :: *Void*

| *WTCall*:

$\llbracket$ $P,E \vdash e :: Class\ C$; $P \vdash C\ sees\ M{:}Ts \to T = (pns,body)\ in\ D$;
$P,E \vdash es\ [::]\ Ts'$; $P \vdash Ts'\ [\le]\ Ts$ $\rrbracket$
$\Longrightarrow P,E \vdash e{\bullet}M(es) :: T$

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

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

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

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

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

— well-typed expression lists

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

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

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

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

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

2.17 Runtime Well-typedness

theory *WellTypeRT*
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* ⊢ *e* : *T* ≡ *WTrt P h E e T*
| *P*,*E*,*h* ⊢ *es*[:]*Ts* ≡ *WTrts P h E es Ts*

| *WTrtNew*:
  is-class P C ⟹
  P,*E*,*h* ⊢ *new C* : *Class C*

| *WTrtCast*:
  ⟦ *P*,*E*,*h* ⊢ *e* : *T*; *is-refT T*; *is-class P C* ⟧
  ⟹ *P*,*E*,*h* ⊢ *Cast C e* : *Class C*

| *WTrtVal*:
  $typeof_h$ *v* = *Some T* ⟹
  P,*E*,*h* ⊢ *Val v* : *T*

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

| *WTrtBinOpEq*:
  ⟦ *P*,*E*,*h* ⊢ e_1 : T_1; *P*,*E*,*h* ⊢ e_2 : T_2 ⟧
  ⟹ *P*,*E*,*h* ⊢ e_1 «*Eq*» e_2 : *Boolean*

| *WTrtBinOpAdd*:
  ⟦ *P*,*E*,*h* ⊢ e_1 : *Integer*; *P*,*E*,*h* ⊢ e_2 : *Integer* ⟧
  ⟹ *P*,*E*,*h* ⊢ e_1 «*Add*» e_2 : *Integer*

| *WTrtLAss*:
  ⟦ *E V* = *Some T*; *P*,*E*,*h* ⊢ *e* : *T′*; *P* ⊢ *T′* ≤ *T* ⟧
  ⟹ *P*,*E*,*h* ⊢ *V*:=*e* : *Void*

| *WTrtFAcc*:
  ⟦ *P*,*E*,*h* ⊢ *e* : *Class C*; *P* ⊢ *C has F*:*T in D* ⟧ ⟹
  P,*E*,*h* ⊢ *e*•*F*{*D*} : *T*

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

2.17.2 Some interesting lemmas

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

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

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

lemma *WTrt-hext-mono*: $P,E,h \vdash e : T \Longrightarrow h \trianglelefteq h' \Longrightarrow P,E,h' \vdash e : T$
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 $hyperUn :: 'a\ hyperset \Rightarrow 'a\ hyperset \Rightarrow 'a\ hyperset$ (**infixl** $\sqcup$ *65*)
where
$A \sqcup B \equiv case\ A\ of\ None \Rightarrow None$
$\quad | \lfloor A \rfloor \Rightarrow (case\ B\ of\ None \Rightarrow None \mid \lfloor B \rfloor \Rightarrow \lfloor A \cup B \rfloor)$

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

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

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

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

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

lemma [*simp*]: $\lfloor\{\}\rfloor \sqcup A = A \ \wedge\ A \sqcup \lfloor\{\}\rfloor = A$
lemma [*simp*]: $\lfloor A\rfloor \sqcup \lfloor B\rfloor = \lfloor A \cup B\rfloor \wedge \lfloor A\rfloor \ominus a = \lfloor A - \{a\}\rfloor$
lemma [*simp*]: $None \sqcup A = None \wedge A \sqcup None = None$
lemma [*simp*]: $a \in\in None \wedge None \ominus a = None$
lemma *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
$\mathcal{A} :: 'a\ exp \Rightarrow 'a\ hyperset$
and $\mathcal{A}s :: 'a\ exp\ list \Rightarrow 'a\ hyperset$
where
$\mathcal{A}\ (new\ C) = \lfloor\{\}\rfloor$
$| \mathcal{A}\ (Cast\ C\ e) = \mathcal{A}\ e$
$| \mathcal{A}\ (Val\ v) = \lfloor\{\}\rfloor$
$| \mathcal{A}\ (e_1 \ll bop \gg e_2) = \mathcal{A}\ e_1 \sqcup \mathcal{A}\ e_2$
$| \mathcal{A}\ (Var\ V) = \lfloor\{\}\rfloor$
$| \mathcal{A}\ (LAss\ V\ e) = \lfloor\{V\}\rfloor \sqcup \mathcal{A}\ e$
$| \mathcal{A}\ (e{\cdot}F\{D\}) = \mathcal{A}\ e$
$| \mathcal{A}\ (e_1{\cdot}F\{D\}{:=}e_2) = \mathcal{A}\ e_1 \sqcup \mathcal{A}\ e_2$

$| \mathcal{A}\ (e\cdot M(es)) = \mathcal{A}\ e \sqcup \mathcal{A}s\ es$
$| \mathcal{A}\ (\{V{:}T;\ e\}) = \mathcal{A}\ e \ominus V$
$| \mathcal{A}\ (e_1;;e_2) = \mathcal{A}\ e_1 \sqcup \mathcal{A}\ e_2$
$| \mathcal{A}\ (\mathit{if}\ (e)\ e_1\ \mathit{else}\ e_2) = \mathcal{A}\ e \sqcup (\mathcal{A}\ e_1 \sqcap \mathcal{A}\ e_2)$
$| \mathcal{A}\ (\mathit{while}\ (b)\ e) = \mathcal{A}\ b$
$| \mathcal{A}\ (\mathit{throw}\ e) = \mathit{None}$
$| \mathcal{A}\ (\mathit{try}\ e_1\ \mathit{catch}(C\ V)\ e_2) = \mathcal{A}\ e_1 \sqcap (\mathcal{A}\ e_2 \ominus V)$

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

primrec
$\mathcal{D}$ :: $'a\ \mathit{exp} \Rightarrow 'a\ \mathit{hyperset} \Rightarrow \mathit{bool}$
and $\mathcal{D}s$ :: $'a\ \mathit{exp\ list} \Rightarrow 'a\ \mathit{hyperset} \Rightarrow \mathit{bool}$
where
$\mathcal{D}\ (\mathit{new}\ C)\ A = \mathit{True}$
$| \mathcal{D}\ (\mathit{Cast}\ C\ e)\ A = \mathcal{D}\ e\ A$
$| \mathcal{D}\ (\mathit{Val}\ v)\ A = \mathit{True}$
$| \mathcal{D}\ (e_1 \ll \mathit{bop} \gg e_2)\ A = (\mathcal{D}\ e_1\ A \wedge \mathcal{D}\ e_2\ (A \sqcup \mathcal{A}\ e_1))$
$| \mathcal{D}\ (\mathit{Var}\ V)\ A = (V \in\in A)$
$| \mathcal{D}\ (\mathit{LAss}\ V\ e)\ A = \mathcal{D}\ e\ A$
$| \mathcal{D}\ (e\cdot F\{D\})\ A = \mathcal{D}\ e\ A$
$| \mathcal{D}\ (e_1\cdot F\{D\}{:=}e_2)\ A = (\mathcal{D}\ e_1\ A \wedge \mathcal{D}\ e_2\ (A \sqcup \mathcal{A}\ e_1))$
$| \mathcal{D}\ (e\cdot M(es))\ A = (\mathcal{D}\ e\ A \wedge \mathcal{D}s\ es\ (A \sqcup \mathcal{A}\ e))$
$| \mathcal{D}\ (\{V{:}T;\ e\})\ A = \mathcal{D}\ e\ (A \ominus V)$
$| \mathcal{D}\ (e_1;;e_2)\ A = (\mathcal{D}\ e_1\ A \wedge \mathcal{D}\ e_2\ (A \sqcup \mathcal{A}\ e_1))$
$| \mathcal{D}\ (\mathit{if}\ (e)\ e_1\ \mathit{else}\ e_2)\ A =$
$(\mathcal{D}\ e\ A \wedge \mathcal{D}\ e_1\ (A \sqcup \mathcal{A}\ e) \wedge \mathcal{D}\ e_2\ (A \sqcup \mathcal{A}\ e))$
$| \mathcal{D}\ (\mathit{while}\ (e)\ c)\ A = (\mathcal{D}\ e\ A \wedge \mathcal{D}\ c\ (A \sqcup \mathcal{A}\ e))$
$| \mathcal{D}\ (\mathit{throw}\ e)\ A = \mathcal{D}\ e\ A$
$| \mathcal{D}\ (\mathit{try}\ e_1\ \mathit{catch}(C\ V)\ e_2)\ A = (\mathcal{D}\ e_1\ A \wedge \mathcal{D}\ e_2\ (A \sqcup \lfloor\{V\}\rfloor))$

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

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

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

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

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

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

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

abbreviation
confs :: $'m\ prog \Rightarrow heap \Rightarrow val\ list \Rightarrow ty\ list \Rightarrow bool$
$(\text{-,-} \vdash \text{-}\ [:\leq]\ \text{-}\ [51,51,51,51]\ 50)$ **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*]: $P,h \vdash empty\ (:\leq)\ E$
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*:
$[\![P,E,h \vdash e : \mathit{Class}\ C;\ \mathit{final}\ e;$
$\bigwedge a.\ e = \mathit{addr}\ a \Longrightarrow R;$
$\bigwedge a.\ e = \mathit{Throw}\ a \Longrightarrow R\ ]\!] \Longrightarrow R$

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

Derivation of new induction scheme for well typing:

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

$\mid \mathit{is\text{-}class}\ P\ C \Longrightarrow P,E,h \vdash \mathit{new}\ C :' \mathit{Class}\ C$
$\mid [\![P,E,h \vdash e :' T;\ \mathit{is\text{-}refT}\ T;\ \mathit{is\text{-}class}\ P\ C]\!]$
$\Longrightarrow P,E,h \vdash \mathit{Cast}\ C\ e :' \mathit{Class}\ C$
$\mid \mathit{typeof}_h\ v = \mathit{Some}\ T \Longrightarrow P,E,h \vdash \mathit{Val}\ v :' T$
$\mid E\ v = \mathit{Some}\ T \Longrightarrow P,E,h \vdash \mathit{Var}\ v :' T$
$\mid [\![P,E,h \vdash e_1 :' T_1;\ P,E,h \vdash e_2 :' T_2]\!]$
$\Longrightarrow P,E,h \vdash e_1 \ll\!\mathit{Eq}\!\gg e_2 :' \mathit{Boolean}$
$\mid [\![P,E,h \vdash e_1 :' \mathit{Integer};\ P,E,h \vdash e_2 :' \mathit{Integer}]\!]$
$\Longrightarrow P,E,h \vdash e_1 \ll\!\mathit{Add}\!\gg e_2 :' \mathit{Integer}$
$\mid [\![P,E,h \vdash \mathit{Var}\ V :' T;\ P,E,h \vdash e :' T';\ P \vdash T' \le T\ (*\ V \ne \mathit{This} *)]\!]$
$\Longrightarrow P,E,h \vdash V{:=}e :' \mathit{Void}$
$\mid [\![P,E,h \vdash e :' \mathit{Class}\ C;\ P \vdash C\ \mathit{has}\ F{:}T\ \mathit{in}\ D]\!] \Longrightarrow P,E,h \vdash e{\cdot}F\{D\} :' T$
$\mid P,E,h \vdash e :' \mathit{NT} \Longrightarrow P,E,h \vdash e{\cdot}F\{D\} :' T$
$\mid [\![P,E,h \vdash e_1 :' \mathit{Class}\ C;\ P \vdash C\ \mathit{has}\ F{:}T\ \mathit{in}\ D;$
$P,E,h \vdash e_2 :' T_2;\ P \vdash T_2 \le T]\!]$
$\Longrightarrow P,E,h \vdash e_1{\cdot}F\{D\}{:=}e_2 :' \mathit{Void}$
$\mid [\![P,E,h \vdash e_1{:}'\mathit{NT};\ P,E,h \vdash e_2 :' T_2]\!] \Longrightarrow P,E,h \vdash e_1{\cdot}F\{D\}{:=}e_2 :' \mathit{Void}$
$\mid [\![P,E,h \vdash e :' \mathit{Class}\ C;\ P \vdash C\ \mathit{sees}\ M{:}Ts \rightarrow T = (\mathit{pns},\mathit{body})\ \mathit{in}\ D;$
$P,E,h \vdash es\ [:']\ Ts';\ P \vdash Ts'\ [\le]\ Ts]\!]$
$\Longrightarrow P,E,h \vdash e{\cdot}M(es) :' T$
$\mid [\![P,E,h \vdash e :' \mathit{NT};\ P,E,h \vdash es\ [:']\ Ts]\!] \Longrightarrow P,E,h \vdash e{\cdot}M(es) :' T$
$\mid P,E,h \vdash []\ [:']\ []$
$\mid [\![P,E,h \vdash e :' T;\ P,E,h \vdash es\ [:']\ Ts]\!] \Longrightarrow P,E,h \vdash e\#es\ [:']\ T\#Ts$
$\mid [\![\mathit{typeof}_h\ v = \mathit{Some}\ T_1;\ P \vdash T_1 \le T;\ P,E(V \mapsto T),h \vdash e_2 :' T_2]\!]$

$\Longrightarrow\ P,E,h \vdash \{V{:}T := Val\ v;\ e_2\} :' T_2$
$\mid [\![\ P,E(V \mapsto T),h \vdash e :' T';\ \neg\ assigned\ V\ e\ ]\!] \Longrightarrow\ P,E,h \vdash \{V{:}T;\ e\} :' T'$
$\mid [\![\ P,E,h \vdash e_1{:}' T_1;\ \ P,E,h \vdash e_2{:}'T_2\ ]\!]\ \Longrightarrow\ P,E,h \vdash e_1;;e_2 :' T_2$
$\mid [\![\ P,E,h \vdash e :' Boolean;\ \ P,E,h \vdash e_1{:}' T_1;\ \ P,E,h \vdash e_2{:}' T_2;$
$P \vdash T_1 \le T_2 \lor P \vdash T_2 \le T_1;$
$P \vdash T_1 \le T_2 \longrightarrow T = T_2;\ P \vdash T_2 \le T_1 \longrightarrow T = T_1\ ]\!]$
$\Longrightarrow P,E,h \vdash if\ (e)\ e_1\ else\ e_2 :' T$

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

lemma [*iff*]: $P,E,h \vdash e_1;;e_2 :' T_2 = (\exists\, T_1.\ P,E,h \vdash e_1{:}' T_1 \land P,E,h \vdash e_2{:}' T_2)$
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*: $P,E,h \vdash e : T \Longrightarrow$
$(\bigwedge l.\ [\![\ \mathcal{D}\ e\ \lfloor dom\ l \rfloor;\ \neg\ final\ e\ ]\!] \Longrightarrow \exists\, e'\ s'.\ P \vdash \langle e,(h,l)\rangle \to \langle e',s'\rangle)$
and $P,E,h \vdash es\ [:]\ Ts \Longrightarrow$
$(\bigwedge l.\ [\![\ \mathcal{D}s\ es\ \lfloor dom\ l \rfloor;\ \neg\ finals\ es\ ]\!] \Longrightarrow \exists\, es'\ s'.\ P \vdash \langle es,(h,l)\rangle\ [\to]\ \langle es',s'\rangle)$

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*) ∧
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*) ∧
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* ⟧
⟹ *wf-J-mdecl P C jmdcl*

lemma *wf-mdecl-wwf-mdecl*: *wf-J-mdecl P C Md* ⟹ *wwf-J-mdecl P C Md*

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

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

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

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

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

Now preservation of definite assignment.

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

Combining conformance of heap and local variables:

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

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

2.22.2 Subject reduction

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

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

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

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

2.22.3 Lifting to $\rightarrow*$

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

lemma *Red-preserves-sconf*:
assumes *wf*: *wf-J-prog P* **and** *Red*: $P \vdash \langle e,s\rangle \rightarrow* \langle e',s'\rangle$
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*: $P \vdash \langle e,s\rangle \rightarrow* \langle e',s'\rangle$
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*: $P \vdash \langle e,s\rangle \rightarrow* \langle e',s'\rangle$
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 $P \vdash \langle e,s\rangle \rightarrow \langle e',s'\rangle \Longrightarrow P,E,s \vdash e : T \surd$
$\Longrightarrow \exists T'.\ P,E,s' \vdash e' : T' \surd \wedge P \vdash T' \leq T$

theorem *Subject-reductions*:
assumes *wf*: *wf-J-prog P* **and** *reds*: $P \vdash \langle e,s\rangle \rightarrow* \langle e',s'\rangle$
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 $[\![\ P,E,s \vdash e : T \surd; \mathcal{D}\ e\ \lfloor dom(lcl\ s)\rfloor; \neg\ final\ e\ ]\!] \Longrightarrow \exists e'\ s'.\ P \vdash \langle e,s\rangle \rightarrow \langle e',s'\rangle$

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

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 ⊢ new C ⇝ new C
| AnnoCast: P,E ⊢ e ⇝ e′ ⟹ P,E ⊢ Cast C e ⇝ Cast C e′
| AnnoVal: P,E ⊢ Val v ⇝ Val v
| AnnoVarVar: E V = ⌊T⌋ ⟹ P,E ⊢ Var V ⇝ Var V
| AnnoVarField: ⟦ E V = None; E this = ⌊Class C⌋; P ⊢ C sees V:T in D ⟧
              ⟹ P,E ⊢ Var V ⇝ Var this·V{D}
| AnnoBinOp:
  ⟦ P,E ⊢ e1 ⇝ e1′;  P,E ⊢ e2 ⇝ e2′ ⟧
  ⟹ P,E ⊢ e1 «bop» e2 ⇝ e1′ «bop» e2′
| AnnoLAssVar:
  ⟦ E V = ⌊T⌋; P,E ⊢ e ⇝ e′ ⟧ ⟹ P,E ⊢ V:=e ⇝ V:=e′
| AnnoLAssField:
  ⟦ E V = None; E this = ⌊Class C⌋; P ⊢ C sees V:T in D; P,E ⊢ e ⇝ e′ ⟧
  ⟹ P,E ⊢ V:=e ⇝ Var this·V{D} := e′
| AnnoFAcc:
  ⟦ P,E ⊢ e ⇝ e′;  P,E ⊢ e′ :: Class C;  P ⊢ C sees F:T in D ⟧
  ⟹ P,E ⊢ e·F{[]} ⇝ e′·F{D}
| AnnoFAss: ⟦ P,E ⊢ e1 ⇝ e1′;  P,E ⊢ e2 ⇝ e2′;
            P,E ⊢ e1′ :: Class C; P ⊢ C sees F:T in D ⟧
          ⟹ P,E ⊢ e1·F{[]} := e2 ⇝ e1′·F{D} := e2′
| AnnoCall:
  ⟦ P,E ⊢ e ⇝ e′;  P,E ⊢ es [⇝] es′ ⟧
  ⟹ P,E ⊢ Call e M es ⇝ Call e′ M es′
| AnnoBlock:
  P,E(V ↦ T) ⊢ e ⇝ e′  ⟹  P,E ⊢ {V:T; e} ⇝ {V:T; e′}
| AnnoComp: ⟦ P,E ⊢ e1 ⇝ e1′;  P,E ⊢ e2 ⇝ e2′ ⟧
          ⟹  P,E ⊢ e1;;e2 ⇝ e1′;;e2′
| AnnoCond: ⟦ P,E ⊢ e ⇝ e′; P,E ⊢ e1 ⇝ e1′;  P,E ⊢ e2 ⇝ e2′ ⟧
          ⟹ P,E ⊢ if (e) e1 else e2 ⇝ if (e′) e1′ else e2′
| AnnoLoop: ⟦ P,E ⊢ e ⇝ e′;  P,E ⊢ c ⇝ c′ ⟧
          ⟹ P,E ⊢ while (e) c ⇝ while (e′) c′
| AnnoThrow: P,E ⊢ e ⇝ e′  ⟹  P,E ⊢ throw e ⇝ throw e′
| AnnoTry: ⟦ P,E ⊢ e1 ⇝ e1′;  P,E(V ↦ Class C) ⊢ e2 ⇝ e2′ ⟧
         ⟹ P,E ⊢ try e1 catch(C V) e2 ⇝ try e1′ catch(C V) e2′

| AnnoNil: P,E ⊢ [] [⇝] []
| AnnoCons: ⟦ P,E ⊢ e ⇝ e′;  P,E ⊢ es [⇝] es′ ⟧
          ⟹  P,E ⊢ e#es [⇝] 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
  ~~/src/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 ⟹ 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 ⊢ ⟨e,s0⟩ ⇒ ⟨null,s1⟩;  P ⊢ ⟨ps,s1⟩ [⇒] ⟨evs,s2⟩; map-val evs vs ⟧
  ⟹ P ⊢ ⟨e•M(ps),s0⟩ ⇒ ⟨THROW NullPointer,s2⟩
apply(rule CallNull, assumption+)
apply(simp add: map-val-conv[symmetric])
done


lemma CallParamsThrow2:
  ⟦ P ⊢ ⟨e,s0⟩ ⇒ ⟨Val v,s1⟩; P ⊢ ⟨es,s1⟩ [⇒] ⟨evs,s2⟩;
     map-val2 evs vs (throw ex # es'') ⟧
   ⟹ P ⊢ ⟨e•M(es),s0⟩ ⇒ ⟨throw ex,s2⟩
apply(rule eval-evals.CallParamsThrow, assumption+)
apply(simp add: map-val2-conv[symmetric])
done

lemma Call2:
  ⟦ P ⊢ ⟨e,s0⟩ ⇒ ⟨addr a,s1⟩;  P ⊢ ⟨ps,s1⟩ [⇒] ⟨evs,(h2,l2)⟩;
     map-val evs vs;
     h2 a = Some(C,fs);  P ⊢ C sees M:Ts→T = (pns,body) in D;
     length vs = length pns;  l2' = [this↦Addr a, pns[↦]vs];
     P ⊢ ⟨body,(h2,l2')⟩ ⇒ ⟨e',(h3,l3)⟩ ⟧
  ⟹ P ⊢ ⟨e•M(ps),s0⟩ ⇒ ⟨e',(h3,l2)⟩
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 CallNull2[OF refl])(simp add: map-val-conv[symmetric])
  next
    case CallParamsThrow thus ?thesis
      by(rule CallParamsThrow2[OF refl])(simp add: map-val2-conv[symmetric])
  next
    case Call thus ?thesis
      by −(rule Call2[OF refl], simp-all add: map-val-conv[symmetric])
  next
    case WhileBodyThrow thus ?thesis by(rule WhileBodyThrow2[OF refl])
  qed(assumption|erule (4) that[OF refl]|erule (3) that[OF refl])+
```

```
next
  case evals
  from evals.prems show thesis
    by(cases (no-simp))(assumption|erule (3) that[OF refl])+
qed

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

definition test1 = [] ⊢ ⟨testExpr1,(empty,empty)⟩ ⇒ ⟨-,-⟩
definition test2 = [] ⊢ ⟨testExpr2,(empty,empty)⟩ ⇒ ⟨-,-⟩
definition test3 = [] ⊢ ⟨testExpr3,(empty,empty(''V''↦Intg 77))⟩ ⇒ ⟨-,-⟩
definition test4 = [] ⊢ ⟨testExpr4,(empty,empty)⟩ ⇒ ⟨-,-⟩
definition test5 = [(''Object'',('''',[],[])),(''C'',(''Object'',[(''F'',Integer)],[]))] ⊢ ⟨testExpr5,(empty,empty)⟩
⇒ ⟨-,-⟩
definition test6 = [(''Object'',('''',[],[])), classI] ⊢ ⟨testExpr6,(empty,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, (empty,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*, (*empty*,*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*: $i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow bool$ *as type-check*, $i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow bool$ *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 ‹$x \vdash T1 \leq T2 \lor x \vdash T2 \leq T1$› **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 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 WTCond2[OF refl])
    qed
  qed(assumption|erule (2) that[OF refl])+
next
  case WTs
  from WTs.prems show thesis
    by(cases (no-simp))(assumption|erule (2) that[OF refl])+
qed

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

definition test1 where test1 = [],empty ⊢ testExpr1 :: -
definition test2 where test2 = [], empty ⊢ testExpr2 :: -
definition test3 where test3 = [], empty(''V'' ↦ Integer) ⊢ testExpr3 :: -
definition test4 where test4 = [], empty(''V'' ↦ Integer) ⊢ testExpr4 :: -
definition test5 where test5 = [classObject, (''C'',(''Object'',[(''F'',Integer)],[]))], empty ⊢ testExpr5
:: -
definition test6 where test6 = [classObject, classI], 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], empty([this] [↦] [Class ''A''])
⊢ mb-isNull :: -
definition testmb-add where testmb-add = [classObject, classA], empty([this,''i''] [↦] [Class ''A'',Integer])
⊢ mb-add :: -
definition testmb-mult-cond where testmb-mult-cond = [classObject, classA], empty([this,''j''] [↦]
[Class ''A'',Integer]) ⊢ mb-mult-cond :: -
definition testmb-mult-block where testmb-mult-block = [classObject, classA], empty([this,''i'',''j'',''temp'']
[↦] [Class ''A'',Integer,Integer,Integer]) ⊢ mb-mult-block :: -
definition testmb-mult where testmb-mult = [classObject, classA], 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*], *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 C0 M0 pc frs =
     (None, h, ((loc ! n) # stk, loc, C0, M0, pc+1)#frs)

| exec-instr (Store n) P h stk loc C0 M0 pc frs =
     (None, h, (tl stk, loc[n:=hd stk], C0, M0, pc+1)#frs)

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

| exec-instr-New:
 exec-instr (New C) P h stk loc C0 M0 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 mxl_0 *undefined),D,M,0)*
in (xp′, h, f′#(stk, loc, C_0*,* M_0*, pc)#frs))*

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

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

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

| *exec-instr (IfFalse i) P h stk loc* C_0 M_0 *pc frs =*
(let pc′ = if hd stk = Bool False then nat(int pc+i) else pc+1
in (None, h, (tl stk, loc, C_0*,* M_0*, pc′)#frs))*

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

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

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

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

end

3.4 Exception handling in the JVM

theory *JVMExceptions* **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* $\preceq^*$ *C'*

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

abbreviation
  ex-table-of :: *jvm-prog* ⇒ *cname* ⇒ *mname* ⇒ *ex-table* **where**
  ex-table-of P C M == *snd* (*snd* (*snd* (*snd* (*snd* (*snd*(*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 (xsymbols)
  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 ⊢ σ −jvm→ σ
lemma jvm-trans[trans]:
 ⟦ P ⊢ σ −jvm→ σ'; P ⊢ σ' −jvm→ σ'' ⟧ ⟹ P ⊢ σ −jvm→ σ''
lemma jvm-one-step1[trans]:
 ⟦ P ⊢ σ −jvm→1 σ'; P ⊢ σ' −jvm→ σ'' ⟧ ⟹ P ⊢ σ −jvm→ σ''
```

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 M₀ 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 =
```

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

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

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

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

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

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

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

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

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

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

definition *check* :: *jvm-prog* ⇒ *jvm-state* ⇒ *bool* **where**

check P σ = (*let* (*xcpt*, *h*, *frs*) = σ *in*
(*case frs of* [] $\Rightarrow$ *True* | (*stk*,*loc*,*C*,*M*,*pc*)#*frs'* $\Rightarrow$
P $\vdash$ *C has M* $\wedge$
(*let* (*C'*,*Ts*,*T*,*mxs*,mxl_0,*ins*,*xt*) = *method P C M*; *i* = *ins*!*pc in*
pc < *size ins* $\wedge$ *size stk* $\leq$ *mxs* $\wedge$
check-instr i P h stk loc C M pc frs')))

definition *exec-d* :: *jvm-prog* $\Rightarrow$ *jvm-state* $\Rightarrow$ *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* $\Rightarrow$ (*jvm-state type-error* $\times$ *jvm-state type-error*) *set*
and *exec-1-d'* :: *jvm-prog* $\Rightarrow$ *jvm-state type-error* $\Rightarrow$ *jvm-state type-error* $\Rightarrow$ *bool*
(- $\vdash$ - $-jvmd\rightarrow_1$ - [61,61,61]60)
for *P* :: *jvm-prog*
where
P $\vdash$ σ $-jvmd\rightarrow_1$ σ' $\equiv$ (σ,σ') $\in$ *exec-1-d P*
| *exec-1-d-ErrorI*: *exec-d P* σ = *TypeError* $\Longrightarrow$ *P* $\vdash$ *Normal* σ $-jvmd\rightarrow_1$ *TypeError*
| *exec-1-d-NormalI*: *exec-d P* σ = *Normal* (*Some* σ') $\Longrightarrow$ *P* $\vdash$ *Normal* σ $-jvmd\rightarrow_1$ *Normal* σ'

— reflexive transitive closure:
definition *exec-all-d* :: *jvm-prog* $\Rightarrow$ *jvm-state type-error* $\Rightarrow$ *jvm-state type-error* $\Rightarrow$ *bool*
(- |− - −*jvmd*−> - [61,61,61]60) **where**
exec-all-d-def1: *P* |− σ −*jvmd*−> σ' $\longleftrightarrow$ (σ,σ') $\in$ (*exec-1-d P*)*

notation (*xsymbols*)
exec-all-d (- $\vdash$ - $-jvmd\rightarrow$ - [61,61,61]60)

lemma *exec-1-d-eq*:
exec-1-d P = {(*s*,*t*). $\exists\sigma$. *s* = *Normal* σ $\wedge$ *t* = *TypeError* $\wedge$ *exec-d P* σ = *TypeError*} $\cup$
{(*s*,*t*). $\exists\sigma\ \sigma'$. *s* = *Normal* σ $\wedge$ *t* = *Normal* σ' $\wedge$ *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*) $\neq$ *B* $\Longrightarrow$ *P*
by (*cases P*, *auto*)

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

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

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

end

3.7 Example for generating executable code from JVM semantics

```
theory JVMListExample
imports
  ../Common/SystemClasses
  JVMExec
  ~~/src/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-const undefined-cname
  (SML object)

definition undefined-val :: val
  where [code del]: undefined-val = undefined
declare undefined-val-def[symmetric, code-unfold]
```

code-const *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);

  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;
⟩⟩
```

end

Chapter 4

Bytecode Verifier

4.1 Semilattices

theory *Semilat*
imports *Main* ~~/*src*/*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*

consts
  lesub :: *'a* ⇒ *'a ord* ⇒ *'a* ⇒ *bool*
  lesssub :: *'a* ⇒ *'a ord* ⇒ *'a* ⇒ *bool*
  plussub :: *'a* ⇒ (*'a* ⇒ *'b* ⇒ *'c*) ⇒ *'b* ⇒ *'c***notation** (*xsymbols*)
  lesub ((- /⊑- -) [*50*, *0*, *51*] *50*) **and**
  lesssub ((- /⊏- -) [*50*, *0*, *51*] *50*) **and**
  plussub ((- /⊔- -) [*65*, *0*, *66*] *65*)
defs
  lesub-def: $x \sqsubseteq_r y \equiv r\ x\ y$
  lesssub-def: $x \sqsubset_r y \equiv x \sqsubseteq_r y \wedge x \neq y$
  plussub-def: $x \sqcup_f y \equiv f\ x\ y$

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

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

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

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

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

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

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

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* $\Longrightarrow x \sqsubseteq_r x$

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

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

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

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

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

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

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

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

lemma (**in** *Semilat*) *closedI* [*simp*, *intro*]: *closed A f*

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

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

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

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

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

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

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

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

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

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

lemma (**in** *Semilat*) *le-iff-plus-unchanged*:
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 $(r\hat{}*)$ *x y y* $= ((x,y) \in r\hat{}*)$
lemma *is-lub-bigger2* [*iff*]:
is-lub $(r\hat{}*)$ *x y x* $= ((y,x) \in r\hat{}*)$
lemma *extend-lub*:
$[\![$ *single-valued r*; *is-lub* $(r\hat{}*)$ *x y u*; $(x',x) \in r$ $]\!]$
$\Longrightarrow$ *EX v*. *is-lub* $(r\hat{}*)$ x' *y v*
lemma *single-valued-has-lubs* [*rule-format*]:
$[\![$ *single-valued r*; $(x,u) \in r\hat{}*$ $]\!] \Longrightarrow (\forall y. (y,u) \in r\hat{}* \longrightarrow$
(*EX z*. *is-lub* $(r\hat{}*)$ *x y z*))
lemma *some-lub-conv*:
$[\![$ *acyclic r*; *is-lub* $(r\hat{}*)$ *x y u* $]\!] \Longrightarrow$ *some-lub* $(r\hat{}*)$ *x y* $= u$
lemma *is-lub-some-lub*:
$[\![$ *single-valued r*; *acyclic r*; $(x,u) \in r\hat{}*$; $(y,u) \in r\hat{}*$ $]\!]$
$\Longrightarrow$ *is-lub* $(r\hat{}*)$ *x y* (*some-lub* $(r\hat{}*)$ *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* $(\lambda z. (x,z) \notin r^*)$ *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*; $(x,u){:}r\hat{}*$; $(y,u){:}r\hat{}*$; $\forall x\ y.\ (x,y) \in r \longrightarrow f\ x = y\ ]\!]$
$\Longrightarrow$ *is-lub* $(r\hat{}*\ )\ x\ y$ (*exec-lub* $r\ f\ x\ y$)

end

4.2 The Error Type

theory *Err*
imports *Semilat*
begin

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

type-synonym $'a\ ebinop = 'a \Rightarrow 'a \Rightarrow 'a\ err$
type-synonym $'a\ esl = 'a\ set \times 'a\ ord \times 'a\ ebinop$

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

definition $lift :: ('a \Rightarrow 'b\ err) \Rightarrow ('a\ err \Rightarrow 'b\ err)$
where
  $lift\ f\ e = (case\ e\ of\ Err \Rightarrow Err \mid OK\ x \Rightarrow f\ x)$

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

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

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

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

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

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

abbreviation
  $err\text{-}semilat :: 'a\ esl \Rightarrow bool$ **where**
  $err\text{-}semilat\ L == semilat(sl\ L)$

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

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

4.2.1 lift

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

4.2.2 sup

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

4.2.3 semilat (err A) (le r) f

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

$[\![\ x\in\ err\ A;\ semilat(err\ A,\ le\ r,\ f)\ ]\!] \Longrightarrow x \sqcup_f\ Err = Err$

lemma *semilat-le-err-OK1*:

$[\![\ x\in A;\ y\in A;\ semilat(err\ A,\ le\ r,\ f);\ OK\ x \sqcup_f\ OK\ y = OK\ z\ ]\!]$

$\Longrightarrow x \sqsubseteq_r z$

lemma *semilat-le-err-OK2*:

$[\![\ x\in A;\ y\in A;\ semilat(err\ A,\ le\ r,\ f);\ OK\ x \sqcup_f\ OK\ y = OK\ z\ ]\!]$

$\Longrightarrow y \sqsubseteq_r z$

lemma *eq-order-le*:

$[\![\ x=y;\ order\ r\ ]\!] \Longrightarrow x \sqsubseteq_r y$

lemma *OK-plus-OK-eq-Err-conv* [*simp*]:

assumes $x\in A\quad y\in A\quad semilat(err\ A,\ le\ r,\ fe)$

shows $(OK\ x \sqcup_{fe}\ OK\ y = Err) = (\neg(\exists z\in A.\ x \sqsubseteq_r z \land y \sqsubseteq_r z))$

4.2.4 semilat (err(Union AS))

lemma *all-bex-swap-lemma* [*iff*]:

$(\forall x.\ (\exists y\in A.\ x = f\ y) \longrightarrow P\ x) = (\forall y\in A.\ P(f\ y))$

lemma *closed-err-Union-lift2I*:

$[\![\ \forall A\in AS.\ closed\ (err\ A)\ (lift2\ f);\ AS \neq \{\};$

$\forall A\in AS.\forall B\in AS.\ A\neq B \longrightarrow (\forall a\in A.\forall b\in B.\ a \sqcup_f\ b = Err)\ ]\!]$

$\Longrightarrow closed\ (err(Union\ AS))\ (lift2\ f)$

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

lemma *err-semilat-UnionI*:

$[\![\ \forall A\in AS.\ err\text{-}semilat(A,\ r,\ f);\ AS \neq \{\};$

$\forall A\in AS.\forall B\in AS.\ A\neq B \longrightarrow (\forall a\in A.\forall b\in B.\ \neg a \sqsubseteq_r b \land a \sqcup_f\ b = Err)\ ]\!]$

$\Longrightarrow err\text{-}semilat(Union\ AS,\ r,\ f)$

end

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* ∈ *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* ⟹ $x \sqsubseteq_{le\ r} x$

4.4 Products as Semilattices

theory *Product*
imports *Err*
begin

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

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

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

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

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

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

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

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

4.5 Fixed Length Lists

theory *Listn*
imports *Err*
begin

definition *list* :: *nat* ⇒ *'a set* ⇒ *'a list set*
where
list n A = {*xs. size xs* = *n* ∧ *set xs* ⊆ *A*}

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

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

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

definition *map2* :: (*'a* ⇒ *'b* ⇒ *'c*) ⇒ *'a list* ⇒ *'b list* ⇒ *'c list*
where
map2 f = (λ*xs ys. map* (*split f*) (*zip xs ys*))

abbreviation (*xsymbols*)
plussublist :: *'a list* ⇒ (*'a* ⇒ *'b* ⇒ *'c*) ⇒ *'b list* ⇒ *'c list*
((- /[⊔-] -) [*65*, *0*, *66*] *65*) **where**
x [$\sqcup_f$] *y* == *x* $\sqcup_{map2\ f}$ *y*

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

definition *sl* :: *nat* ⇒ *'a sl* ⇒ *'a list sl*
where
sl n = (λ(*A*,*r*,*f*). (*list 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* [$\sqcup_f$] *ys*) *else Err*)

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

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

lemma *unfold-lesub-list*: *xs* [$\sqsubseteq_r$] *ys* = *Listn.le r xs ys*
lemma *Nil-le-conv* [*iff*]: ([] [$\sqsubseteq_r$] *ys*) = (*ys* = [])

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

lemma *le-listD*: $\llbracket\ xs\ [\sqsubseteq_r]\ ys;\ p < size\ xs\ \rrbracket \Longrightarrow xs!p \sqsubseteq_r ys!p$
lemma *le-list-refl*: $\forall x.\ x \sqsubseteq_r x \Longrightarrow xs\ [\sqsubseteq_r]\ xs$
lemma *le-list-trans*: $\llbracket\ order\ r;\ xs\ [\sqsubseteq_r]\ ys;\ ys\ [\sqsubseteq_r]\ zs\ \rrbracket \Longrightarrow xs\ [\sqsubseteq_r]\ zs$
lemma *le-list-antisym*: $\llbracket\ order\ r;\ xs\ [\sqsubseteq_r]\ ys;\ ys\ [\sqsubseteq_r]\ xs\ \rrbracket \Longrightarrow xs = ys$
lemma *order-listI* [*simp*, *intro*!]: $order\ r \Longrightarrow order(Listn.le\ r)$
lemma *lesub-list-impl-same-size* [*simp*]: $xs\ [\sqsubseteq_r]\ ys \Longrightarrow size\ ys = size\ xs$
lemma *lesssub-lengthD*: $xs\ [\sqsubset_r]\ ys \Longrightarrow size\ ys = size\ xs$
lemma *le-list-appendI*: $a\ [\sqsubseteq_r]\ b \Longrightarrow c\ [\sqsubseteq_r]\ d \Longrightarrow a@c\ [\sqsubseteq_r]\ b@d$
lemma *le-listI*:
  assumes $length\ a = length\ b$
  assumes $\bigwedge n.\ n < length\ a \Longrightarrow a!n \sqsubseteq_r b!n$
  shows $a\ [\sqsubseteq_r]\ b$
lemma *listI*: $\llbracket\ size\ xs = n;\ set\ xs \subseteq A\ \rrbracket \Longrightarrow xs \in list\ n\ A$

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

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

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

lemma (**in** *Semilat*) *plus-list-ub1* [*rule-format*]:
$[\![$ *set xs* $\subseteq A$; *set ys* $\subseteq A$; *size xs* = *size ys* $]\!]$
$\Longrightarrow$ *xs* $[\sqsubseteq_r]$ *xs* $[\sqcup_f]$ *ys*
lemma (**in** *Semilat*) *plus-list-ub2*:
$[\![$*set xs* $\subseteq A$; *set ys* $\subseteq A$; *size xs* = *size ys* $]\!] \Longrightarrow$ *ys* $[\sqsubseteq_r]$ *xs* $[\sqcup_f]$ *ys*
lemma (**in** *Semilat*) *plus-list-lub* [*rule-format*]:
shows $\forall$ *xs ys zs*. *set xs* $\subseteq A \longrightarrow$ *set ys* $\subseteq A \longrightarrow$ *set zs* $\subseteq A$
$\longrightarrow$ *size xs* = *n* $\wedge$ *size ys* = *n* $\longrightarrow$
xs $[\sqsubseteq_r]$ *zs* $\wedge$ *ys* $[\sqsubseteq_r]$ *zs* $\longrightarrow$ *xs* $[\sqcup_f]$ *ys* $[\sqsubseteq_r]$ *zs*
lemma (**in** *Semilat*) *list-update-incr* [*rule-format*]:
$x \in A \Longrightarrow$ *set xs* $\subseteq A \longrightarrow$
($\forall$ *i*. *i*<*size xs* $\longrightarrow$ *xs* $[\sqsubseteq_r]$ *xs*[*i* := *x* $\sqcup_f$ *xs*!*i*])
lemma *acc-le-listI* [*intro*!]:
$[\![$ *order r*; *acc r* $]\!] \Longrightarrow$ *acc*(*Listn.le r*)
lemma *closed-listI*:
closed S f $\Longrightarrow$ *closed* (*list n S*) (*map2 f*)

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

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

$map2\ f\ xs\ ys \in list\ n\ (err\ A))$

lemma *closed-lift2-sup*:

$closed\ (err\ A)\ (lift2\ f) \Longrightarrow$

$closed\ (err\ (list\ n\ A))\ (lift2\ (sup\ f))$

lemma *err-semilat-sup*:

$err\text{-}semilat\ (A,r,f) \Longrightarrow$

$err\text{-}semilat\ (list\ n\ A,\ Listn.le\ r,\ sup\ f)$

lemma *err-semilat-upto-esl*:

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

end

4.6 Typing and Dataflow Analysis Framework

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

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

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

definition *stable* $:: 's\ ord \Rightarrow 's\ step\text{-}type \Rightarrow 's\ list \Rightarrow nat \Rightarrow bool$
where
$stable\ r\ step\ \tau s\ p \longleftrightarrow (\forall (q,\tau) \in set\ (step\ p\ (\tau s!p)).\ \tau \sqsubseteq_r \tau s!q)$

definition *stables* $:: 's\ ord \Rightarrow 's\ step\text{-}type \Rightarrow 's\ list \Rightarrow bool$
where
$stables\ r\ step\ \tau s \longleftrightarrow (\forall p < size\ \tau s.\ stable\ r\ step\ \tau s\ p)$

definition *wt-step* $:: 's\ ord \Rightarrow 's \Rightarrow 's\ step\text{-}type \Rightarrow 's\ list \Rightarrow bool$
where
$wt\text{-}step\ r\ T\ step\ \tau s \longleftrightarrow (\forall p{<}size\ \tau s.\ \tau s!p \neq T \wedge stable\ r\ step\ \tau s\ p)$

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

end

4.7 More on Semilattices

theory *SemilatAlg*
imports *Typing-Framework*
begin

consts
lesubstep-type :: $(nat \times 's)$ *set* $\Rightarrow$ *'s ord* $\Rightarrow$ $(nat \times 's)$ *set* $\Rightarrow$ *bool***notation** (*xsymbols*)
lesubstep-type ((- /{$\sqsubseteq$-} -) [50, 0, 51] 50)
defs *lesubstep-type-def*:
$A \{\sqsubseteq_r\} B \equiv \forall (p,\tau) \in A.\ \exists \tau'.\ (p,\tau') \in B \wedge \tau \sqsubseteq_r \tau'$

primrec *pluslussub* :: *'a list* $\Rightarrow$ ('a $\Rightarrow$ 'a $\Rightarrow$ 'a) $\Rightarrow$ 'a $\Rightarrow$ 'a
where
pluslussub [] *f y* = *y*
| *pluslussub* (*x#xs*) *f y* = *pluslussub xs f* $(x \sqcup_f y)$**notation** (*xsymbols*)
pluslussub ((- /$\bigsqcup$- -) [65, 0, 66] 65)

definition *bounded* :: *'s step-type* $\Rightarrow$ *nat* $\Rightarrow$ *bool*
where
bounded step n $\longleftrightarrow$ $(\forall p<n.\ \forall \tau.\ \forall (q,\tau') \in set\ (step\ p\ \tau).\ q<n)$

definition *pres-type* :: *'s step-type* $\Rightarrow$ *nat* $\Rightarrow$ *'s set* $\Rightarrow$ *bool*
where
pres-type step n A $\longleftrightarrow$ $(\forall \tau \in A.\ \forall p<n.\ \forall (q,\tau') \in set\ (step\ p\ \tau).\ \tau' \in A)$

definition *mono* :: *'s ord* $\Rightarrow$ *'s step-type* $\Rightarrow$ *nat* $\Rightarrow$ *'s set* $\Rightarrow$ *bool*
where
mono r step n A $\longleftrightarrow$
$(\forall \tau\ p\ \tau'.\ \tau \in A \wedge p<n \wedge \tau \sqsubseteq_r \tau' \longrightarrow set\ (step\ p\ \tau)\ \{\sqsubseteq_r\}\ set\ (step\ p\ \tau'))$

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

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

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

lemma *pres-typeD*:
$[\![$ *pres-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** *Typing-Framework SemilatAlg* **begin**

definition *wt-err-step* :: $'s$ *ord* $\Rightarrow$ $'s$ *err step-type* $\Rightarrow$ $'s$ *err list* $\Rightarrow$ *bool*
where
wt-err-step r step τs $\longleftrightarrow$ *wt-step* (*Err.le r*) *Err step* τs

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

definition *map-snd* :: $('b \Rightarrow 'c) \Rightarrow ('a \times 'b)\ list \Rightarrow ('a \times 'c)\ list$
where
map-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-type* $\Rightarrow$ $'s$ *err step-type*
where
err-step n app step p t =
(*case t of*
Err $\Rightarrow$ *error n*
| *OK* τ $\Rightarrow$ *if app p* τ *then map-snd OK* (*step p* τ) *else error n*)

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

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

lemma *bounded-err-stepD*:
$[\![$ *bounded* (*err-step n app step*) *n*;
$p < n$; *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-step n ap step*) *n*

lemma *bounded-lift*:
bounded step n $\Longrightarrow$ *bounded* (*err-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-snd OK xs*) $\{\sqsubseteq_{Err.le\ r}\}$ *set* (*map-snd OK ys*)

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

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

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

4.9 Kildall's Algorithm

theory *Kildall*
imports *SemilatAlg*
begin

primrec *propa* :: $'s\ binop \Rightarrow (nat \times 's)\ list \Rightarrow 's\ list \Rightarrow nat\ set \Rightarrow 's\ list * nat\ set$
where
$propa\ f\ []\quad \tau s\ w = (\tau s, w)$
$|\ propa\ f\ (q'\#qs)\ \tau s\ w = (let\ (q,\tau) = q';$
$\qquad u = \tau \sqcup_f \tau s!q;$
$\qquad w' = (if\ u = \tau s!q\ then\ w\ else\ insert\ q\ w)$
$\qquad in\ propa\ f\ qs\ (\tau s[q := u])\ w')$

definition *iter* :: $'s\ binop \Rightarrow 's\ step\text{-}type \Rightarrow$
$\quad 's\ list \Rightarrow nat\ set \Rightarrow 's\ list \times nat\ set$
where
$iter\ f\ step\ \tau s\ w =$
$\ while\ (\lambda(\tau s,w).\ w \neq \{\})$
$\quad (\lambda(\tau s,w).\ let\ p = SOME\ p.\ p \in w$
$\qquad in\ propa\ f\ (step\ p\ (\tau s!p))\ \tau s\ (w-\{p\}))$
$\quad (\tau s,w)$

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

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

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

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

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

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

lemma (**in** *Semilat*) *merges-preserves-type* [*simp*]:
$[\![$ *xs* $\in$ *list n A*; $\forall\,(p,x) \in$ *set ps*. $p<n \wedge x\in A$ $]\!]$
$\Longrightarrow$ *merges f ps xs* $\in$ *list n A*
by (*simp add*: *merges-preserves-type-lemma*)

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

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

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

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

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

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

iter f step ss0 w0 = (*ss'*,*w'*)
$\longrightarrow$
ss' ∈ *list n A* ∧ *stables r step ss'* ∧ *ss0* $[\sqsubseteq_r]$ *ss'* ∧
(∀ *ts*∈*list n A*. *ss0* $[\sqsubseteq_r]$ *ts* ∧ *stables r step ts* ⟶ *ss'* $[\sqsubseteq_r]$ *ts*)

lemma *kildall-properties*: **assumes** *Semilat A r f*
shows ⟦ *acc r*; *pres-type step n A*; *mono r step n A*;
bounded step n; *ss0* ∈ *list n A* ⟧ ⟹
kildall r f step ss0 ∈ *list n A* ∧
stables r step (*kildall r f step ss0*) ∧
ss0 $[\sqsubseteq_r]$ *kildall r f step ss0* ∧
(∀ *ts*∈*list n A*. *ss0* $[\sqsubseteq_r]$ *ts* ∧ *stables r step ts* ⟶
kildall r f step ss0 $[\sqsubseteq_r]$ *ts*)
end

4.10 The Lightweight Bytecode Verifier

theory *LBVSpec*
imports *SemilatAlg Opt*
begin

type-synonym
  's certificate = 's list

primrec *merge* :: *'s certificate ⇒ 's binop ⇒ 's ord ⇒ 's ⇒ nat ⇒ (nat × 's) list ⇒ 's ⇒ '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*
    if pc'=pc+1 then s' $\sqcup_f$ x
    else if s' $\sqsubseteq_r$ cert!pc' then x
    else T)

definition *wtl-inst* :: *'s certificate ⇒ 's binop ⇒ 's ord ⇒ 's ⇒*
    's step-type ⇒ nat ⇒ 's ⇒ 's
where
  wtl-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 $\sqsubseteq_r$ 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 $\sqsubseteq_r$ 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* $\bot$
assumes *B-A*: $\bot \in A$

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

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

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

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

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

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

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

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

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

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

declare *Let-def* [*simp*]

4.10.1 more semilattice lemmas

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

lemma (**in** *Semilat*) *pp-ub1'*:
assumes *S*: *snd'set S* $\subseteq$ *A*
assumes *y*: $y \in A$ **and** *ab*: (*a*, *b*) $\in$ *set S*
shows *b* $\sqsubseteq_r$ *map snd* [(*p'*, *t'*) $\leftarrow$ *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*]: $x \sqsubseteq_r \bot = (x = \bot)$

by (*blast intro*: *antisym-r*)

4.10.2 merge

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

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

lemma (**in** *lbv*) *merge-Err* [*simp*]:
snd‘*set ss* ⊆ *A* ⟹ *merge c pc ss* ⊤ = ⊤
by (*induct ss*) *auto*

lemma (**in** *lbv*) *merge-not-top*:
⋀*x*. *snd*‘*set ss* ⊆ *A* ⟹ *merge c pc ss x* ≠ ⊤ ⟹
∀(*pc*′,*s*′) ∈ *set ss*. (*pc*′ ≠ *pc*+*1* ⟶ *s*′ ⊑ᵣ *c*!*pc*′)
(**is** ⋀*x*. *?set ss* ⟹ *?merge ss x* ⟹ *?P ss*)

lemma (**in** *lbv*) *merge-def*:
shows
⋀*x*. *x* ∈ *A* ⟹ *snd*‘*set ss* ⊆ *A* ⟹
merge c pc ss x =
(*if* ∀(*pc*′,*s*′) ∈ *set ss*. *pc*′≠*pc*+*1* ⟶ *s*′ ⊑ᵣ *c*!*pc*′ *then*
map snd [(*p*′,*t*′) ← *ss*. *p*′=*pc*+*1*] ⨆_*f* *x*
else ⊤)
(**is** ⋀*x*. - ⟹ - ⟹ *?merge ss x* = *?if ss x* **is** ⋀*x*. - ⟹ - ⟹ *?P ss x*)
lemma (**in** *lbv*) *merge-not-top-s*:
assumes *x*: *x* ∈ *A* **and** *ss*: *snd*‘*set ss* ⊆ *A*
assumes *m*: *merge c pc ss x* ≠ ⊤
shows *merge c pc ss x* = (*map snd* [(*p*′,*t*′) ← *ss*. *p*′=*pc*+*1*] ⨆_*f* *x*)

4.10.3 wtl-inst-list

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

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

lemma (**in** *lbv*) *wtl-Cons* [*simp*]:
wtl (*i*#*is*) *c pc s* =
(*let s*′ = *wtc c pc s in if s*′ = ⊤ ∨ *s* = ⊤ *then* ⊤ *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* ≠ ⊤ =
(*wtc c pc s* ≠ ⊤ ∧ *s* ≠ *T* ∧ *wtl is c* (*pc*+*1*) (*wtc c pc s*) ≠ ⊤)
by (*auto simp del*: *split-paired-Ex*)

lemma (**in** *lbv*) *wtl-top* [*simp*]: *wtl ls c pc* ⊤ = ⊤
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** $?wtl\ is \neq \text{-} \Longrightarrow \text{-}$)
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** $?wtl\ is \neq \text{-}$)
assumes *pc*: $pc < length\ is$
shows $wtc\ c\ pc\ (wtl\ (take\ pc\ is)\ c\ 0\ s) \neq \top$

4.10.4 preserves-type

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

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

4.11 Correctness of the LBV

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

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

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

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

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

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 $\exists \tau s \in list\ (size\ ins)\ A.\ wt\text{-}step\ r\ \top\ step\ \tau s \wedge s_0 \sqsubseteq_r \tau s!0$
end

4.12 Completeness of the LBV

theory *LBVComplete*
imports *LBVSpec Typing-Framework*
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* (τs!*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* τs!*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* (τs!*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. τs!*pc* ∈ *A* ∧ τs!*pc* ≠ ⊤
assumes *bounded*: *bounded step* (*size* τs)

assumes *B-neq-T*: ⊥ ≠ ⊤

lemma (**in** *lbvc*) *cert*: *cert-ok c* (*size* τs) ⊤ ⊥ *A*
lemmas [*simp del*] = *split-paired-Ex*

lemma (**in** *lbvc*) *cert-target* [*intro?*]:
⟦ (*pc'*,*s'*) ∈ *set* (*step pc* (τs!*pc*));
pc' ≠ *pc+1*; *pc* < *size* τs; *pc'* < *size* τs ⟧
⟹ *c*!*pc'* = τs!*pc'*
lemma (**in** *lbvc*) *cert-approx* [*intro?*]:
⟦ *pc* < *size* τs; *c*!*pc* ≠ ⊥ ⟧ ⟹ *c*!*pc* = τs!*pc*
lemma (**in** *lbv*) *le-top* [*simp*, *intro*]: *x* <=-*r* ⊤
lemma (**in** *lbv*) *merge-mono*:
assumes *less*: *set* ss_2 {$\sqsubseteq_r$} *set* ss_1
assumes *x*: *x* ∈ *A*
assumes ss_1: *snd*‘*set* ss_1 ⊆ *A*
assumes ss_2: *snd*‘*set* ss_2 ⊆ *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*]: ⊤ $\sqsubseteq_r$ *x* = (*x* = ⊤)
lemma (**in** *lbv*) *neq-top* [*simp*, *elim*]: ⟦ *x* $\sqsubseteq_r$ *y*; *y* ≠ ⊤ ⟧ ⟹ *x* ≠ ⊤

lemma (**in** *lbvc*) *stable-wti*:
assumes *stable*: *stable r step* τs *pc* **and** *pc*: $pc < size\ \tau s$
shows *wti c pc* $(\tau s!pc) \neq \top$
lemma (**in** *lbvc*) *wti-less*:
assumes *stable*: *stable r step* τs *pc* **and** *suc-pc*: $Suc\ pc < size\ \tau s$
shows *wti c pc* $(\tau s!pc) \sqsubseteq_r \tau s!Suc\ pc$ (**is** $?wti \sqsubseteq_r$ -)
lemma (**in** *lbvc*) *stable-wtc*:
assumes *stable*: *stable r step* τs *pc* **and** *pc*: $pc < size\ \tau s$
shows *wtc c pc* $(\tau s!pc) \neq \top$
lemma (**in** *lbvc*) *wtc-less*:
assumes *stable*: *stable r step* τs *pc* **and** *suc-pc*: $Suc\ pc < size\ \tau s$
shows *wtc c pc* $(\tau s!pc) \sqsubseteq_r \tau s!Suc\ pc$ (**is** $?wtc \sqsubseteq_r$ -)
lemma (**in** *lbvc*) *wt-step-wtl-lemma*:
assumes *wt-step*: *wt-step r* $\top$ *step* τs
shows $\bigwedge pc\ s.\ pc+size\ ls = size\ \tau s \Longrightarrow s \sqsubseteq_r \tau s!pc \Longrightarrow s \in A \Longrightarrow s\neq\top \Longrightarrow$
$wtl\ ls\ c\ pc\ s \neq \top$
(**is** $\bigwedge pc\ s.$ - $\Longrightarrow$ - $\Longrightarrow$ - $\Longrightarrow$ - $\Longrightarrow$ $?wtl\ ls\ pc\ s \neq$ -)
theorem (**in** *lbvc*) *wtl-complete*:
assumes *wt*: *wt-step r* $\top$ *step* τs
assumes *s*: $s \sqsubseteq_r \tau s!0\ s \in A\ s \neq \top$ **and** *eq*: $size\ ins = size\ \tau s$
shows *wtl ins c 0 s* $\neq \top$
end

4.13 The Jinja Type System as a Semilattice

theory *SemiType*
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 = (λT_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* $\preceq^*$ *Object*

lemma *subcls-antisym*:
  ⟦*wf-prog m P*; *P* ⊢ *C* $\preceq^*$ *D*; *P* ⊢ *D* $\preceq^*$ *C*⟧ ⟹ *C* = *D*

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

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*) $\Longrightarrow$ *acc* (*subtype P*)
lemma *wf-subtype-acc* [*intro*, *simp*]:
wf-prog wf-mb P $\Longrightarrow$ *acc* (*subtype P*)
lemma *exec-lub-refl* [*simp*]: *exec-lub r f T T* = *T*
lemma *closed-err-types*:
wf-prog wf-mb P $\Longrightarrow$ *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* $]\!]$
$\Longrightarrow$ *subtype P t1 s* $\wedge$ *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* $]\!]$
$\Longrightarrow$ *subtype P d c*
lemma *sup-exists*:
$[\![$ *subtype P a c*; *subtype P b c* $]\!] \Longrightarrow$ *EX T*. *sup P a b* = *OK T*
lemma *err-semilat-JType-esl*:
wf-prog wf-mb P $\Longrightarrow$ *err-semilat* (*esl P*)

end

4.14 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*
(- |– - <=T - [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 :: $['c\ prog, ty_l, ty_l] \Rightarrow bool$ (- |− - [<=T] - [71,71,71] 70)
where P |− LT [<=T] $LT' \equiv$ *list-all2* (*sup-ty-opt* P) $LT\ LT'$

notation (*xsymbols*)
sup-ty-opt (- $\vdash$ - $\leq_\top$ - [71,71,71] 70) **and**
sup-state (- $\vdash$ - $\leq_i$ - [71,71,71] 70) **and**
sup-state-opt (- $\vdash$ - $\leq'$ - [71,71,71] 70) **and**
sup-loc (- $\vdash$ - $[\leq_\top]$ - [71,71,71] 70)

4.14.1 Unfolding

lemma *JVM-states-unfold*:
states P mxs mxl $\equiv$ *err*(*opt*((*Union* {*list n* (*types P*) |*n*. *n* <= *mxs*}) <*>
list mxl (*err*(*types P*))))
lemma *JVM-le-unfold*:
le P m n $\equiv$
Err.le(*Opt.le*(*Product.le*(*Listn.le*(*subtype P*))(*Listn.le*(*Err.le*(*subtype P*)))))
lemma *sl-def2*:
JVM-SemiType.sl P mxs mxl $\equiv$
(*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*) = $P \vdash t1 \leq' t2$
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*

4.14.2 Semilattice

lemma *order-sup-state-opt* [*intro*, *simp*]:
wf-prog wf-mb P $\Longrightarrow$ *order* (*sup-state-opt P*)
lemma *semilat-JVM* [*intro?*]:
wf-prog wf-mb P $\Longrightarrow$ *semilat* (*JVM-SemiType.sl P mxs mxl*)
lemma *acc-JVM* [*intro*]:
wf-prog wf-mb P $\Longrightarrow$ *acc* (*JVM-SemiType.le P mxs mxl*)

4.14.3 Widening with $\top$

lemma *subtype-refl*[*iff*]: *subtype P t t*
lemma *sup-ty-opt-refl* [*iff*]: $P \vdash T \leq_\top T$
lemma *Err-any-conv* [*iff*]: $P \vdash Err \leq_\top T = (T = Err)$
lemma *any-Err* [*iff*]: $P \vdash T \leq_\top Err$
lemma *OK-OK-conv* [*iff*]:
$P \vdash OK\ T \leq_\top OK\ T' = P \vdash T \leq T'$
lemma *any-OK-conv* [*iff*]:
$P \vdash X \leq_\top OK\ T' = (\exists T.\ X = OK\ T \wedge P \vdash T \leq T')$
lemma *OK-any-conv*:
$P \vdash OK\ T \leq_\top X = (X = Err \vee (\exists T'.\ X = OK\ T' \wedge P \vdash T \leq T'))$
lemma *sup-ty-opt-trans* [*intro?*, *trans*]:
$[\![P \vdash a \leq_\top b; P \vdash b \leq_\top c]\!] \Longrightarrow P \vdash a \leq_\top c$

4.14.4 Stack and Registers

lemma *stk-convert*:

$P \vdash ST\ [\le]\ ST' = Listn.le\ (subtype\ P)\ ST\ ST'$
lemma *sup-loc-refl* [*iff*]: $P \vdash LT\ [\le_\top]\ LT$
lemmas *sup-loc-Cons1* [*iff*] = *list-all2-Cons1* [*of sup-ty-opt P*] **for** *P*

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

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

4.14.5 State Type

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

lemma *sup-state-opt-None-any* [*iff*]:
$P \vdash None \le' s$
lemma *sup-state-opt-any-None* [*iff*]:
$P \vdash s \le' None = (s = None)$
lemma *sup-state-opt-Some-Some* [*iff*]:
$P \vdash Some\ a \le' Some\ b = P \vdash a \le_i b$
lemma *sup-state-opt-any-Some*:
$P \vdash (Some\ s) \le' X = (\exists s'.\ X = Some\ s' \wedge P \vdash s \le_i s')$
lemma *sup-state-opt-refl* [*iff*]: $P \vdash s \le' s$
lemma *sup-state-opt-trans* [*intro?*, *trans*]:
$[\![P \vdash a \le' b;\ P \vdash b \le' c]\!] \Longrightarrow P \vdash a \le' c$
end

4.15 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 mxl0 :: nat
  fixes Ts :: ty list
  fixes Tr :: ty
  fixes is :: instr list
  fixes xt :: ex-table

  fixes mxl :: nat
  defines mxl-def: mxl ≡ 1+size Ts+mxl0
```

Program counter of successor instructions:

```
primrec succs :: instr ⇒ tyi ⇒ 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 effi :: instr × 'm prog × tyi ⇒ tyi where
  effi-Load:
    effi (Load n,  P, (ST, LT))          = (ok-val (LT ! n) # ST, LT)
| effi-Store:
    effi (Store n, P, (T#ST, LT))        = (ST, LT[n:= OK T])
| effi-Push:
```

```
    effᵢ (Push v, P, (ST, LT))            = (the (typeof v) # ST, LT)
| effᵢ-Getfield:
    effᵢ (Getfield F C, P, (T#ST, LT))    = (snd (field P C F) # ST, LT)
| effᵢ-Putfield:
   effᵢ (Putfield F C, P, (T₁#T₂#ST, LT)) = (ST,LT)
| effᵢ-New:
   effᵢ (New C, P, (ST,LT))               = (Class C # ST, LT)
| effᵢ-Checkcast:
   effᵢ (Checkcast C, P, (T#ST,LT))       = (Class C # ST,LT)
| effᵢ-Pop:
   effᵢ (Pop, P, (T#ST,LT))               = (ST,LT)
| effᵢ-IAdd:
   effᵢ (IAdd, P,(T₁#T₂#ST,LT))           = (Integer#ST,LT)
| effᵢ-CmpEq:
   effᵢ (CmpEq, P, (T₁#T₂#ST,LT))         = (Boolean#ST,LT)
| effᵢ-IfFalse:
   effᵢ (IfFalse b, P, (T₁#ST,LT))        = (ST,LT)
| effᵢ-Invoke:
   effᵢ (Invoke M n, P, (ST,LT))          =
    (let C = the-class (ST!n); (D,Ts,Tᵣ,b) = method P C M
     in (Tᵣ # drop (n+1) ST, LT))
| effᵢ-Goto:
   effᵢ (Goto n, P, s)                    = s

fun is-relevant-class :: instr ⇒ 'm prog ⇒ cname ⇒ bool where
  rel-Getfield:
    is-relevant-class (Getfield F D) = (λP C. P ⊢ NullPointer ⪯* C)
| rel-Putfield:
    is-relevant-class (Putfield F D) = (λP C. P ⊢ NullPointer ⪯* C)
| rel-Checcast:
    is-relevant-class (Checkcast D)  = (λP C. P ⊢ ClassCast ⪯* C)
| rel-New:
    is-relevant-class (New D)        = (λP C. P ⊢ OutOfMemory ⪯* C)
| rel-Throw:
    is-relevant-class Throw          = (λP C. True)
| rel-Invoke:
    is-relevant-class (Invoke M n)   = (λP C. True)
| rel-default:
    is-relevant-class i              = (λP 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 ⇒ tyᵢ
               ⇒ ex-table ⇒ (pc × tyᵢ') 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 ⇒ tyᵢ ⇒ (pc × tyᵢ') list where
  norm-eff i P pc τ = map (λpc'. (pc',Some (effᵢ (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* $\wedge$
  (*ST*!$n \neq$ *NT* $\longrightarrow$
   ($\exists$ *C D Ts T m*. *ST*!n = *Class C* $\wedge$ *P* $\vdash$ *C sees M*:*Ts* $\rightarrow$ *T* = *m in D* $\wedge$
         P $\vdash$ *rev* (*take n ST*) $[\leq]$ *Ts*)))

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

definition *xcpt-app* :: *instr* $\Rightarrow$ *'m prog* $\Rightarrow$ *pc* $\Rightarrow$ *nat* $\Rightarrow$ *ex-table* $\Rightarrow$ ty_i $\Rightarrow$ *bool* **where**
 xcpt-app i P pc mxs xt $\tau \longleftrightarrow$ ($\forall$ (*f*,*t*,*C*,*h*,*d*) $\in$ *set* (*relevant-entries P i pc xt*). *is-class P C* $\wedge$ $d \leq$ *size* (*fst* τ) $\wedge$ $d <$ *mxs*)

definition *app* :: *instr* $\Rightarrow$ *'m prog* $\Rightarrow$ *nat* $\Rightarrow$ *ty* $\Rightarrow$ *nat* $\Rightarrow$ *nat* $\Rightarrow$ *ex-table* $\Rightarrow$ ty_i' $\Rightarrow$ *bool* **where**
 app i P mxs T_r *pc mpc xt t* = (*case t of None* $\Rightarrow$ *True* | *Some* $\tau \Rightarrow$
 app_i (*i*,*P*,*pc*,*mxs*,T_r,τ) $\wedge$ *xcpt-app i P pc mxs xt* τ $\wedge$
 ($\forall$ (*pc'*,τ') $\in$ *set* (*eff i P pc xt t*). *pc'* $<$ *mpc*))

lemma *app-Some*:
 app i P mxs T_r *pc mpc xt* (*Some* τ) =
 (app_i (*i*,*P*,*pc*,*mxs*,T_r,τ) $\wedge$ *xcpt-app i P pc mxs xt* τ $\wedge$
 ($\forall$ (*pc'*,*s'*) $\in$ *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* $\equiv$ *size is*

 defines eff_i *i* $\tau \equiv$ *Effect*.eff_i (*i*,*P*,τ)
 notes eff_i-*simps* [*simp*] = *Effect*.eff_i.*simps* [**where** *P* = *P*, *folded* eff_i-*def*]

 defines app_i *i pc* $\tau \equiv$ *Effect*.app_i (*i*, *P*, *pc*, *mxs*, T_r, τ)
 notes app_i-*simps* [*simp*] = *Effect*.app_i.*simps* [**where** *P*=*P* **and** *mxs*=*mxs* **and** T_r=T_r, *folded* app_i-*def*]

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

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

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

 defines *xcpt-app i pc* $\tau \equiv$ *Effect*.*xcpt-app i P pc mxs xt* τ

notes *xcpt-app* = *Effect.xcpt-app-def* [*of - P - mxs xt*, *folded xcpt-app-def*]

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

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

lemma *length-cases3*:
assumes $\bigwedge LT.\ P\ ([],LT)$
assumes $\bigwedge l\ LT.\ P\ ([l],LT)$
assumes $\bigwedge l\ ST\ LT.\ P\ (l\#ST,LT)$
shows *P s*
lemma *length-cases4*:
assumes $\bigwedge LT.\ P\ ([],LT)$
assumes $\bigwedge l\ LT.\ P\ ([l],LT)$
assumes $\bigwedge l\ l'\ LT.\ P\ ([l,l'],LT)$
assumes $\bigwedge l\ l'\ ST\ LT.\ P\ (l\#l'\#ST,LT)$
shows *P s*

simp rules for *app*

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

lemma *appLoad*[*simp*]:
app_i (*Load idx*, *P*, T_r, *mxs*, *pc*, *s*) = (∃ *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*) =
(∃ *ST LT*. *s* = (*ST*,*LT*) ∧ *length ST* < *mxs* ∧ *typeof v* ≠ *None*)
by (*cases s*, *simp*)

lemma *appGetField*[*simp*]:
app_i (*Getfield F C*,*P*,*pc*,*mxs*,T_r,*s*) =
(∃ *oT vT ST LT*. *s* = (*oT*#*ST*, *LT*) ∧
P ⊢ *C sees F*:*vT in C* ∧ *P* ⊢ *oT* ≤ (*Class C*))
by (*rule length-cases2* [*of - s*]) *auto*

lemma *appPutField*[*simp*]:
app_i (*Putfield F C*,*P*,*pc*,*mxs*,T_r,*s*) =
(∃ *vT vT' oT ST LT*. *s* = (*vT*#*oT*#*ST*, *LT*) ∧
P ⊢ *C sees F*:*vT' in C* ∧ *P* ⊢ *oT* ≤ (*Class C*) ∧ *P* ⊢ *vT* ≤ *vT'*)

by (*rule length-cases4* [*of - s*], *auto*)

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

lemma *appCheckcast*[*simp*]:
app_i (*Checkcast C*,*P*,*pc*,*mxs*,T_r,*s*) =
(∃ *T ST LT*. *s* = (*T*#*ST*,*LT*) ∧ *is-class P C* ∧ *is-refT T*)
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*) = (∃ *ts ST LT*. *s* = (*ts*#*ST*,*LT*))
by (*rule length-cases2*, *auto*)

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

lemma *appIfFalse* [*simp*]:
app_i (*IfFalse b*,*P*,*pc*,*mxs*,T_r,*s*) =
(∃ *ST LT*. *s* = (*Boolean*#*ST*,*LT*) ∧ *0* ≤ *int pc* + *b*)
lemma *appCmpEq*[*simp*]:
app_i (*CmpEq*,*P*,*pc*,*mxs*,T_r,*s*) =
(∃ T_1 T_2 *ST LT*. *s* = (T_1#T_2#*ST*,*LT*) ∧ (¬*is-refT* T_1 ∧ T_2 = T_1 ∨ *is-refT* T_1 ∧ *is-refT* T_2))
by (*rule length-cases4*, *auto*)

lemma *appReturn*[*simp*]:
app_i (*Return*,*P*,*pc*,*mxs*,T_r,*s*) = (∃ *T ST LT*. *s* = (*T*#*ST*,*LT*) ∧ *P* ⊢ *T* ≤ T_r)
by (*rule length-cases2*, *auto*)

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

lemma *effNone*:
(*pc*′, *s*′) ∈ *set* (*eff i P pc et None*) ⟹ *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

4.16 Monotonicity of eff and app

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

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

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

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

lemma eff_i*-mono*:
  assumes *wf*: *wf-prog p P*
  assumes *less*: $P \vdash \tau \leq_i \tau'$
  assumes app_i: $app\ i\ P\ m\ rT\ pc\ mpc\ xt\ (Some\ \tau')$
  assumes *succs*: $succs\ i\ \tau\ pc \neq []\ \ succs\ i\ \tau'\ pc \neq []$
  shows $P \vdash eff_i\ (i,P,\tau) \leq_i eff_i\ (i,P,\tau')$
end

4.17 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*

notation (*input*)
  wf-jvm-prog-phi (*wf′-jvm′-prog*$_{-}$ - [*0*,*999*] *1000*)

lemma *wt-jvm-progD*:
  $wf\text{-}jvm\text{-}prog_{\Phi}\ P \Longrightarrow \exists wt.\ wf\text{-}prog\ wt\ P$

lemma *wt-jvm-prog-impl-wt-instr*:
  $[\![\ wf\text{-}jvm\text{-}prog_{\Phi}\ P;$
    $P \vdash C\ sees\ M{:}Ts \rightarrow T = (mxs,mxl_0,ins,xt)\ in\ C;\ pc < size\ ins\ ]\!]$
  $\Longrightarrow P,T,mxs,size\ ins,xt \vdash ins!pc,pc :: \Phi\ C\ M$

lemma *wt-jvm-prog-impl-wt-start*:
  $[\![\ wf\text{-}jvm\text{-}prog_{\Phi}\ P;$
    $P \vdash C\ sees\ M{:}Ts \rightarrow T = (mxs,mxl_0,ins,xt)\ in\ C\ ]\!] \Longrightarrow$
  $0 < size\ ins \wedge wt\text{-}start\ P\ C\ Ts\ mxl_0\ (\Phi\ C\ M)$

end

4.18 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 ⇒ tyᵢ' err step-type
where
  exec G maxs rT et bs ≡
  err-step (size bs) (λpc. app (bs!pc) G maxs rT pc (size bs) et)
                     (λpc. eff (bs!pc) G pc et)

locale JVM-sl =
  fixes P :: jvm-prog and mxs and mxl₀
  fixes Ts :: ty list and is and xt and Tᵣ

  fixes mxl and A and r and f and app and eff and step
  defines [simp]: mxl ≡ 1+size Ts+mxl₀
  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 ≡ λpc. Effect.app (is!pc) P mxs Tᵣ pc (size is) xt
  defines [simp]: eff ≡ λpc. Effect.eff (is!pc) P pc xt
  defines [simp]: step ≡ err-step (size is) app eff


locale start-context = JVM-sl +
  fixes p and C
  assumes wf: wf-prog p P
  assumes C:  is-class P C
  assumes Ts: set Ts ⊆ types P

  fixes first :: tyᵢ' and start
  defines [simp]:
  first ≡ Some ([],OK (Class C) # map OK Ts @ replicate mxl₀ Err)
  defines [simp]:
  start ≡ OK first # replicate (size is − 1) (OK None)
```

4.18.1 Connecting JVM and Framework

```
lemma (in JVM-sl) step-def-exec: step ≡ exec P mxs Tᵣ xt is
  by (simp add: exec-def)

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

lemma ex-in-list [iff]:
  (∃n. ST ∈ list n A ∧ n ≤ mxs) = (set ST ⊆ A ∧ size ST ≤ mxs)
  by (unfold list-def) auto

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

by *auto*

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

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

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

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

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

lemma *lesubstep-type-simple*:
xs $[\sqsubseteq_{Product.le\ (op\ =)\ r}]$ *ys* ⟹ *set xs* $\{\sqsubseteq_r\}$ *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* $\sqsubseteq_{sup\text{-}state\text{-}opt\ P}$ *t*; *app pc t*⟧
⟹ *set* (*eff pc s*) $\{\sqsubseteq_{sup\text{-}state\text{-}opt\ P}\}$ *set* (*eff pc t*)
lemma (**in** *JVM-sl*) *bounded-step*: *bounded step* (*size is*)
theorem (**in** *JVM-sl*) *step-mono*:
wf-prog wf-mb P ⟹ *mono r step* (*size is*) *A*

lemma (**in** *start-context*) *first-in-A* [*iff*]: *OK first* ∈ *A*
using *Ts C* **by** (*force intro*!: *list-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 mxl_0 *τs* ∧
wt-app-eff (*sup-state-opt P*) *app eff τs*)

end

4.19 Kildall for the JVM

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

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

definition *wt-kildall* :: *jvm-prog* ⇒ *cname* ⇒ *ty list* ⇒ *ty* ⇒ *nat* ⇒ *nat* ⇒
instr list ⇒ *ex-table* ⇒ *bool*
where
wt-kildall P C′ Ts T_r *mxs* mxl_0 *is xt* ≡
0 < *size is* ∧
(*let first* = *Some* ([],[*OK* (*Class C′*)]@(*map OK Ts*)@(*replicate* mxl_0 *Err*));
start = *OK first*#(*replicate* (*size is* − *1*) (*OK None*));
result = *kiljvm P mxs* (*1*+*size Ts*+mxl_0) T_r *is xt start*
in ∀ *n* < *size is*. *result*!*n* ≠ *Err*)

definition *wf-jvm-prog*$_k$ :: *jvm-prog* ⇒ *bool*
where
wf-jvm-prog$_k$ *P* ≡
wf-prog (λ*P C′* (*M*,*Ts*,T_r,(*mxs*,mxl_0,*is*,*xt*)). *wt-kildall P C′ Ts* T_r *mxs* mxl_0 *is xt*) *P*

theorem (**in** *start-context*) *is-bcv-kiljvm*:
is-bcv r Err step (*size is*) *A* (*kiljvm P mxs mxl* T_r *is xt*)

lemma *subset-replicate* [*intro?*]: *set* (*replicate n x*) ⊆ {*x*}
by (*induct n*) *auto*

lemma *in-set-replicate*:
assumes *x* ∈ *set* (*replicate n y*)
shows *x* = *y*
lemma (**in** *start-context*) *start-in-A* [*intro?*]:
0 < *size is* ⟹ *start* ∈ *list* (*size is*) *A*
using *Ts C*

theorem (**in** *start-context*) *wt-kil-correct*:
assumes *wtk*: *wt-kildall P C Ts* T_r *mxs* mxl_0 *is xt*
shows ∃τ*s*. *wt-method P C Ts* T_r *mxs* mxl_0 *is xt* τ*s*

theorem (**in** *start-context*) *wt-kil-complete*:
assumes *wtm*: *wt-method P C Ts* T_r *mxs* mxl_0 *is xt* τ*s*
shows *wt-kildall P C Ts* T_r *mxs* mxl_0 *is xt*

theorem *jvm-kildall-correct*:
wf-jvm-prog$_k$ *P* = *wf-jvm-prog P*
end

4.20 LBV for the JVM

theory *LBVJVM*
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* ∈ *list* (*size is*) *A*. *wt-step r Err step* τ*s* ∧ *OK first* $\sqsubseteq_r$ τ*s*!*0*

lemma (**in** *start-context*) *wt-lbv-wt-method*:
assumes *lbv*: *wt-lbv P C Ts* T_r *mxs* mxl_0 *xt cert is*
shows $\exists \tau s$. *wt-method P C Ts* T_r *mxs* mxl_0 *is xt* τs

lemma (**in** *start-context*) *wt-method-wt-lbv*:
assumes *wt*: *wt-method P C Ts* T_r *mxs* mxl_0 *is xt* τs
defines [*simp*]: *cert* $\equiv$ *mk-cert P mxs* T_r *xt is* τs

shows *wt-lbv P C Ts* T_r *mxs* mxl_0 *xt cert is*

theorem *jvm-lbv-correct*:
wt-jvm-prog-lbv P Cert $\Longrightarrow$ *wf-jvm-prog P*
theorem *jvm-lbv-complete*:
assumes *wt*: *wf-jvm-prog*$_\Phi$ *P*
shows *wt-jvm-prog-lbv P* (*prg-cert P* Φ)
end

4.21 BV Type Safety Invariant

```
theory BVConform
imports BVSpec ../JVM/JVMExec ../Common/Conform
begin


definition confT :: 'c prog ⇒ heap ⇒ val ⇒ ty err ⇒ bool
          (-,- |− - :<=T - [51,51,51,51] 50)
where
  P,h |− v :<=T E ≡ case E of Err ⇒ True | OK T ⇒ P,h ⊢ v :≤ T

notation (xsymbols)
  confT  (-,- ⊢ - :≤_T - [51,51,51,51] 50)

abbreviation
  confTs :: 'c prog ⇒ heap ⇒ val list ⇒ ty_l ⇒ bool
          (-,- |− - [:<=T] - [51,51,51,51] 50) where
  P,h |− vs [:<=T] Ts ≡ list-all2 (confT P h) vs Ts

notation (xsymbols)
  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 [:≤_T] 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 [:≤_T] 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
                (-,- |− - [ok]  [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 (*xsymbols*)
correct-state (-,- ⊢ - √ [*61*,*0*,*0*] *61*)

4.21.1 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* ⊢ *x* :≤⊤ *X* = (*X* = *Err* ∨ (∃ *T*. *X* = *OK T* ∧ *P*,*h* ⊢ *x* :≤ *T*))
by (*cases X*) *auto*

lemma *confT-hext* [*intro?*, *trans*]:
⟦ *P*,*h* ⊢ *x* :≤⊤ *T*; *h* ⊴ *h'* ⟧ ⟹ *P*,*h'* ⊢ *x* :≤⊤ *T*
by (*cases T*) (*blast intro*: *conf-hext*)+

lemma *confT-widen* [*intro?*, *trans*]:
⟦ *P*,*h* ⊢ *x* :≤⊤ *T*; *P* ⊢ *T* ≤⊤ *T'* ⟧ ⟹ *P*,*h* ⊢ *x* :≤⊤ *T'*
by (*cases T'*, *auto intro*: *conf-widen*)

4.21.2 Stack and Registers

lemmas *confTs-Cons1* [*iff*] = *list-all2-Cons1* [*of confT P h*] **for** *P h*

lemma *confTs-confT-sup*:
⟦ *P*,*h* ⊢ *loc* [:≤⊤] *LT*; *n* < *size LT*; *LT*!*n* = *OK T*; *P* ⊢ *T* ≤ *T'* ⟧
⟹ *P*,*h* ⊢ (*loc*!*n*) :≤ *T'*
lemma *confTs-hext* [*intro?*]:
P,*h* ⊢ *loc* [:≤⊤] *LT* ⟹ *h* ⊴ *h'* ⟹ *P*,*h'* ⊢ *loc* [:≤⊤] *LT*
by (*fast elim*: *list-all2-mono confT-hext*)

lemma *confTs-widen* [*intro?*, *trans*]:
P,*h* ⊢ *loc* [:≤⊤] *LT* ⟹ *P* ⊢ *LT* [≤⊤] *LT'* ⟹ *P*,*h* ⊢ *loc* [:≤⊤] *LT'*
by (*rule list-all2-trans*, *rule confT-widen*)

lemma *confTs-map* [*iff*]:
⋀*vs*. (*P*,*h* ⊢ *vs* [:≤⊤] *map OK Ts*) = (*P*,*h* ⊢ *vs* [:≤] *Ts*)
by (*induct Ts*) (*auto simp add*: *list-all2-Cons2*)

lemma *reg-widen-Err* [*iff*]:
⋀*LT*. (*P* ⊢ *replicate n Err* [≤⊤] *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*

4.21.3 correct-frames

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

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

4.22 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.

4.22.1 Preliminaries

Simp and intro setup for the type safety proof:

lemmas *defs1* = *correct-state-def conf-f-def wt-instr-def eff-def norm-eff-def app-def xcpt-app-def*

lemmas *widen-rules* [*intro*] = *conf-widen confT-widen confs-widens confTs-widen*

4.22.2 Exception Handling

For the *Invoke* instruction the BV has checked all handlers that guard the current *pc*.

lemma *Invoke-handlers*:
  match-ex-table P C pc xt = *Some* (*pc′*,*d′*) $\Longrightarrow$
  ∃(*f*,*t*,*D*,*h*,*d*) ∈ *set* (*relevant-entries P* (*Invoke n M*) *pc xt*).
   P ⊢ *C* $\preceq^*$ *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: *split-if-asm*)

We can prove separately that the recursive search for exception handlers (*find-handler*) in the frame stack results in a conforming state (if there was no matching exception handler in the current frame). We require that the exception is a valid heap address, and that the state before the exception occured conforms.

term *find-handler*
lemma *uncaught-xcpt-correct*:
  assumes *wt*: *wf-jvm-prog*$_\Phi$ *P*
  assumes *h*: *h xcp* = *Some obj*
  shows ⋀*f*. *P*,Φ ⊢ (*None*, *h*, *f*#*frs*)√ $\Longrightarrow$ *P*,Φ ⊢ (*find-handler P xcp h frs*) √
  (**is** ⋀*f*. *?correct* (*None*, *h*, *f*#*frs*) $\Longrightarrow$ *?correct* (*?find frs*))

The requirement of lemma *uncaught-xcpt-correct* (that the exception is a valid reference on the heap) is always met for welltyped instructions and conformant states:

lemma *exec-instr-xcpt-h*:
  ⟦ *fst* (*exec-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*)√ ⟧
  $\Longrightarrow$ ∃ *obj*. *h xcp* = *Some obj*
  (**is** ⟦ *?xcpt*; *?wt*; *?correct* ⟧ $\Longrightarrow$ *?thesis*)
lemma *conf-sys-xcpt*:
  ⟦*preallocated h*; *C* ∈ *sys-xcpts*⟧ $\Longrightarrow$ *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′*) $\Longrightarrow$
  ∃(*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*: *split-if-asm*)

Finally we can state that, whenever an exception occurs, the next state always conforms:

lemma *xcpt-correct*:
fixes σ' :: *jvm-state*
assumes *wtp*: $\textit{wf-jvm-prog}_\Phi$ *P*
assumes *meth*: $P \vdash C$ *sees* $M{:}Ts {\rightarrow} T{=}(mxs,mxl_0,ins,xt)$ *in C*
assumes *wt*: $P,T,mxs,size\ ins,xt \vdash ins!pc,pc :: \Phi\ C\ M$
assumes *xp*: *fst (exec-instr (ins!pc) P h stk loc C M pc frs) = Some xcp*
assumes *s'*: *Some* σ' = *exec (P, None, h, (stk,loc,C,M,pc)#frs)*
assumes *correct*: $P,\Phi \vdash$ *(None, h, (stk,loc,C,M,pc)#frs)*$\surd$
shows $P,\Phi \vdash \sigma'\surd$

4.22.3 Single Instructions

In this section we prove for each single (welltyped) instruction that the state after execution of the instruction still conforms. Since we have already handled exceptions above, we can now assume that no exception occurs in this step.

declare *defs1* [*simp*]

lemma *Invoke-correct*:
fixes σ' :: *jvm-state*
assumes *wtprog*: $\textit{wf-jvm-prog}_\Phi$ *P*
assumes *meth-C*: $P \vdash C$ *sees* $M{:}Ts {\rightarrow} T{=}(mxs,mxl_0,ins,xt)$ *in C*
assumes *ins*: *ins ! pc = Invoke M' n*
assumes *wti*: $P,T,mxs,size\ ins,xt \vdash ins!pc,pc :: \Phi\ C\ M$
assumes σ': *Some* σ' = *exec (P, None, h, (stk,loc,C,M,pc)#frs)*
assumes *approx*: $P,\Phi \vdash$ *(None, h, (stk,loc,C,M,pc)#frs)*$\surd$
assumes *no-xcp*: *fst (exec-instr (ins!pc) P h stk loc C M pc frs) = None*
shows $P,\Phi \vdash \sigma'\surd$
declare *list-all2-Cons2* [*iff*]

lemma *Return-correct*:
fixes σ' :: *jvm-state*
assumes *wt-prog*: $\textit{wf-jvm-prog}_\Phi$ *P*
assumes *meth*: $P \vdash C$ *sees* $M{:}Ts {\rightarrow} T{=}(mxs,mxl_0,ins,xt)$ *in C*
assumes *ins*: *ins ! pc = Return*
assumes *wt*: $P,T,mxs,size\ ins,xt \vdash ins!pc,pc :: \Phi\ C\ M$
assumes *s'*: *Some* σ' = *exec (P, None, h, (stk,loc,C,M,pc)#frs)*
assumes *correct*: $P,\Phi \vdash$ *(None, h, (stk,loc,C,M,pc)#frs)*$\surd$

shows $P,\Phi \vdash \sigma'\surd$
declare *sup-state-opt-any-Some* [*iff*]
declare *not-Err-eq* [*iff*]

lemma *Load-correct*:
⟦ *wf-prog wt P*;
$P \vdash C$ *sees* $M{:}Ts {\rightarrow} T{=}(mxs,mxl_0,ins,xt)$ *in C*;
ins!pc = Load idx;
$P,T,mxs,size\ ins,xt \vdash ins!pc,pc :: \Phi\ C\ M$;
Some σ' = *exec (P, None, h, (stk,loc,C,M,pc)#frs)*;
$P,\Phi \vdash$ *(None, h, (stk,loc,C,M,pc)#frs)*$\surd$ ⟧
$\Longrightarrow P,\Phi \vdash \sigma'\surd$
by (*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*;
  $P \vdash C$ *sees* $M{:}Ts{\rightarrow}T{=}(mxs,mxl_0,ins,xt)$ *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*;
  $P \vdash C$ *sees* $M{:}Ts{\rightarrow}T{=}(mxs,mxl_0,ins,xt)$ *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*:
⟦ $wf\text{-}jvm\text{-}prog_\Phi$ *P*;
  $P \vdash C$ *sees* $M{:}Ts{\rightarrow}T{=}(mxs,mxl_0,ins,xt)$ *in* C;
  ins!*pc* = *Checkcast D*;
  P,*T*,*mxs*,*size ins*,*xt* ⊢ *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*: $P \vdash C$ *sees* $M{:}Ts{\rightarrow}T{=}(mxs,mxl_0,ins,xt)$ *in* C
  assumes *i*: *ins*!*pc* = *Getfield F D*
  assumes *wt*: *P*,*T*,*mxs*,*size ins*,*xt* ⊢ *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*: $P \vdash C$ *sees* $M{:}Ts{\rightarrow}T{=}(mxs,mxl_0,ins,xt)$ *in* C
  assumes *i*: *ins*!*pc* = *Putfield F D*

```
  assumes wt: P,T,mxs,size ins,xt ⊢ 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 has-fields-b-fields:
  P ⊢ C has-fields FDTs ⟹ fields P C = FDTs

lemma oconf-blank [intro, simp]:
    ⟦is-class P C; wf-prog wt P⟧ ⟹ P,h ⊢ blank P C √
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 ⊢ C sees M:Ts→T=(mxs,mxl0,ins,xt) in C
  assumes ins:  ins!pc = New X
  assumes wt:   P,T,mxs,size ins,xt ⊢ ins!pc,pc :: Φ C M
  assumes exec: Some σ' = exec (P, None, h, (stk,loc,C,M,pc)#frs)
  assumes conf: P,Φ ⊢ (None, h, (stk,loc,C,M,pc)#frs)√
  assumes no-x: fst (exec-instr (ins!pc) P h stk loc C M pc frs) = None
  shows P,Φ ⊢ σ'√

lemma Goto-correct:
⟦ wf-prog wt P;
  P ⊢ C sees M:Ts→T=(mxs,mxl0,ins,xt) in C;
  ins ! pc = Goto branch;
  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 IfFalse-correct:
⟦ wf-prog wt P;
  P ⊢ C sees M:Ts→T=(mxs,mxl0,ins,xt) in C;
  ins ! pc = IfFalse branch;
  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 CmpEq-correct:
⟦ wf-prog wt P;
  P ⊢ C sees M:Ts→T=(mxs,mxl0,ins,xt) in C;
  ins ! pc = CmpEq;
  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 Pop-correct:
⟦ wf-prog wt P;
  P ⊢ C sees M:Ts→T=(mxs,mxl0,ins,xt) in C;
```

ins ! pc = Pop;
$P,T,mxs,size\ ins,xt \vdash ins!pc,pc :: \Phi\ C\ M$;
$Some\ \sigma' = exec\ (P,\ None,\ h,\ (stk,loc,C,M,pc)\#frs)$;
$P,\Phi \vdash (None,\ h,\ (stk,loc,C,M,pc)\#frs)\surd$ $]\!]$
$\Longrightarrow P,\Phi \vdash \sigma'\surd$

lemma *IAdd-correct*:
$[\![$ *wf-prog wt P*;
$P \vdash C\ sees\ M{:}Ts{\rightarrow}T{=}(mxs,mxl_0,ins,xt)\ in\ C$;
ins ! pc = $IAdd$;
$P,T,mxs,size\ ins,xt \vdash ins!pc,pc :: \Phi\ C\ M$;
$Some\ \sigma' = exec\ (P,\ None,\ h,\ (stk,loc,C,M,pc)\#frs)$;
$P,\Phi \vdash (None,\ h,\ (stk,loc,C,M,pc)\#frs)\surd$ $]\!]$
$\Longrightarrow P,\Phi \vdash \sigma'\surd$

lemma *Throw-correct*:
$[\![$ *wf-prog wt P*;
$P \vdash C\ sees\ M{:}Ts{\rightarrow}T{=}(mxs,mxl_0,ins,xt)\ in\ C$;
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-instr (ins!pc) P h stk loc C M pc frs) = None $]\!]$
$\Longrightarrow P,\Phi \vdash \sigma'\surd$
by *simp*

The next theorem collects the results of the sections above, i.e. exception handling and the execution step for each instruction. It states type safety for single step execution: in welltyped programs, a conforming state is transformed into another conforming state when one instruction is executed.

theorem *instr-correct*:
$[\![$ *wf-jvm-prog*$_\Phi$ *P*;
$P \vdash C\ sees\ M{:}Ts{\rightarrow}T{=}(mxs,mxl_0,ins,xt)\ in\ C$;
$Some\ \sigma' = exec\ (P,\ None,\ h,\ (stk,loc,C,M,pc)\#frs)$;
$P,\Phi \vdash (None,\ h,\ (stk,loc,C,M,pc)\#frs)\surd$ $]\!]$
$\Longrightarrow P,\Phi \vdash \sigma'\surd$

4.22.4 Main

lemma *correct-state-impl-Some-method*:
$P,\Phi \vdash (None,\ h,\ (stk,loc,C,M,pc)\#frs)\surd$
$\Longrightarrow \exists\, m\ Ts\ T.\ P \vdash C\ sees\ M{:}Ts{\rightarrow}T = m\ in\ C$
by *fastforce*

lemma *BV-correct-1* [*rule-format*]:
$\bigwedge\sigma.\ [\![$ *wf-jvm-prog*$_\Phi$ $P;\ P,\Phi \vdash \sigma\surd]\!] \Longrightarrow P \vdash \sigma\ {-jvm\rightarrow_1}\ \sigma' \longrightarrow P,\Phi \vdash \sigma'\surd$

theorem *progress*:
$[\![\ xp{=}None;\ frs{\neq}[]\ ]\!] \Longrightarrow \exists\,\sigma'.\ P \vdash (xp,h,frs)\ {-jvm\rightarrow_1}\ \sigma'$
by (*clarsimp simp add*: *exec-1-iff neq-Nil-conv split-beta*
simp del: *split-paired-Ex*)

lemma *progress-conform*:
$[\![$*wf-jvm-prog*$_\Phi$ $P;\ P,\Phi \vdash (xp,h,frs)\surd;\ xp{=}None;\ frs{\neq}[]]\!]$
$\Longrightarrow \exists\,\sigma'.\ P \vdash (xp,h,frs)\ {-jvm\rightarrow_1}\ \sigma' \wedge P,\Phi \vdash \sigma'\surd$

theorem *BV-correct* [*rule-format*]:
$[\![$ *wf-jvm-prog*$_\Phi$ P; $P \vdash \sigma$ *−jvm→* σ' $]\!] \Longrightarrow P,\Phi \vdash \sigma\surd \longrightarrow P,\Phi \vdash \sigma'\surd$
lemma *hconf-start*:
  assumes *wf*: *wf-prog wf-mb P*
  shows $P \vdash$ (*start-heap P*) $\surd$
lemma *BV-correct-initial*:
  shows $[\![$ *wf-jvm-prog*$_\Phi$ P; $P \vdash C$ *sees* M:[]→T = m *in* C $]\!]$
  $\Longrightarrow P,\Phi \vdash$ *start-state P C M* $\surd$
theorem *typesafe*:
  assumes *welltyped*: *wf-jvm-prog*$_\Phi$ P
  assumes *main-method*: $P \vdash C$ *sees* M:[]→T = m *in* C
  shows $P \vdash$ *start-state P C M* *−jvm→* $\sigma \Longrightarrow P,\Phi \vdash \sigma \surd$
end

4.23 Welltyped Programs produce no Type Errors

theory *BVNoTypeError*
imports *../JVM/JVMDefensive BVSpecTypeSafe*
begin

lemma *has-methodI*:
$P \vdash C$ *sees* $M{:}Ts{\rightarrow}T = m$ *in* $D \Longrightarrow P \vdash C$ *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*]: *typeof* $v = $ *Some* $x \Longrightarrow \neg$*is-Addr* v
by (*cases* v) *auto*

lemma *is-Ref-def2*:
is-Ref $v = (v = $ *Null* $\vee (\exists a.\ v = $ *Addr* $a))$
by (*cases* v) (*auto simp add*: *is-Ref-def*)

lemma [*iff*]: *is-Ref Null* **by** (*simp add*: *is-Ref-def2*)

lemma *is-RefI* [*intro*, *simp*]: $P,h \vdash v :\leq T \Longrightarrow$ *is-refT* $T \Longrightarrow$ *is-Ref* v
lemma *is-IntgI* [*intro*, *simp*]: $P,h \vdash v :\leq$ *Integer* $\Longrightarrow$ *is-Intg* v
lemma *is-BoolI* [*intro*, *simp*]: $P,h \vdash v :\leq$ *Boolean* $\Longrightarrow$ *is-Bool* v
declare *defs1* [*simp del*]

lemma *wt-jvm-prog-states*:
$[\![$ *wf-jvm-prog*$_\Phi$ P; $P \vdash C$ *sees* M: $Ts{\rightarrow}T = (mxs, mxl, ins, et)$ *in* C;
$\Phi\ C\ M\ !\ pc = \tau$; $pc <$ *size ins* $]\!]$
$\Longrightarrow OK\ \tau \in$ *states* P *mxs* $(1{+}$*size* $Ts{+}mxl)$

The main theorem: welltyped programs do not produce type errors if they are started in a conformant state.

theorem *no-type-error*:
fixes σ :: *jvm-state*
assumes *welltyped*: *wf-jvm-prog*$_\Phi$ P **and** *conforms*: $P,\Phi \vdash \sigma\ \surd$
shows *exec-d* $P\ \sigma \neq$ *TypeError*

The theorem above tells us that, in welltyped programs, the defensive machine reaches the same result as the aggressive one (after arbitrarily many steps).

theorem *welltyped-aggressive-imp-defensive*:
wf-jvm-prog$_\Phi$ $P \Longrightarrow P,\Phi \vdash \sigma\ \surd \Longrightarrow P \vdash \sigma$ *−jvm→* σ'
$\Longrightarrow P \vdash$ (*Normal* σ) *−jvmd→* (*Normal* σ')

As corollary we get that the aggressive and the defensive machine are equivalent for welltyped programs (if started in a conformant state or in the canonical start state)

corollary *welltyped-commutes*:
fixes σ :: *jvm-state*
assumes *wf*: *wf-jvm-prog*$_\Phi$ P **and** *conforms*: $P,\Phi \vdash \sigma\ \surd$
shows $P \vdash$ (*Normal* σ) *−jvmd→* (*Normal* σ') $= P \vdash \sigma$ *−jvm→* σ'
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 ⊢ C sees M:[]→T = b in C
  defines start: σ ≡ start-state P C M
  shows P ⊢ (Normal σ) −jvmd→ (Normal σ') = P ⊢ σ −jvm→ σ'
proof −
  from wf obtain Φ where wf': wf-jvm-prog_Φ P by (auto simp: wf-jvm-prog-def)
  from this meth have P,Φ ⊢ σ √ unfolding start by (rule BV-correct-initial)
  with wf' show ?thesis by (rule welltyped-commutes)
qed


lemma not-TypeError-eq [iff]:
  x ≠ TypeError = (∃ t. x = Normal t)
  by (cases x) auto

locale cnf =
  fixes P and Φ and σ
  assumes wf: wf-jvm-prog_Φ P
  assumes cnf: correct-state P Φ σ

theorem (in cnf) no-type-errors:
  P ⊢ (Normal σ) −jvmd→ σ' ⟹ σ' ≠ 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 ⊢ C sees M:[]→T = b in C
  defines σ ≡ Normal (start-state P C M)

corollary (in start) bv-no-type-error:
  shows P ⊢ σ −jvmd→ σ' ⟹ σ' ≠ TypeError
proof −
  from wf obtain Φ where wf-jvm-prog_Φ 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 ⊢ σ −jvmd→ σ'
  ultimately show ?thesis by (unfold σ-def) (rule cnf.no-type-errors)
qed
```

end

4.24 Example Welltypings

theory *BVExample*
imports *../JVM/JVMListExample BVSpecTypeSafe BVExec*
~~/src/HOL/Library/Code-Target-Numeral
begin

This theory shows type correctness of the example program in section 3.7 (p. 92) 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.

4.24.1 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*:
$(a,b) \in r^* \Longrightarrow a \neq b \Longrightarrow (\bigwedge y.\ (a,y) \notin r) \Longrightarrow False$
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*

4.24.2 Program structure

The program is structurally wellformed:

lemma *wf-struct*:
wf-prog (λ*G C mb. True*) *E* (**is** *wf-prog ?mb E*)

4.24.3 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
φ_a ≡ *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
$x \in [a,\ b) \equiv a \leq x \wedge x < b$

lemma *pc-0*: $x < n \Longrightarrow (x \in [0,\ n) \Longrightarrow P\ x) \Longrightarrow P\ x$
by (*simp add*: *intervall-def*)

lemma *pc-next*: $x \in [n0,\ n) \Longrightarrow P\ n0 \Longrightarrow (x \in [Suc\ n0,\ n) \Longrightarrow P\ x) \Longrightarrow P\ x$
lemma *pc-end*: $x \in [n,n) \Longrightarrow P\ x$
by (*unfold intervall-def*) *arith*

lemma *types-append* [*simp*]: *check-types E 3* (*Suc* (*Suc 0*)) (*map OK* φ_a)
lemma *wt-append* [*simp*]:
wt-method E list-name [*Class list-name*] *Void 3 0 append-ins*
[(*Suc 0*, *2*, *NullPointer*, *7*, *0*)] φ_a

Some abbreviations for readability

abbreviation *Clist* == *Class list-name*
abbreviation *Ctest* == *Class test-name*

definition *phi-makelist* :: ty_m (φ_m)
where
φ_m ≡ *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*;
    $\bigwedge$*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*:
  $wf\text{-}jvm\text{-}prog_{\Phi}$ *E*

4.24.4 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* $\surd$

4.24.5 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 $\bigwedge$*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 *op* ∪ ∘ (λ*p*. *if* ¬ *stable r step ss p then* {*p*} *else* {}) =
      (λ*p 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-const *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*))
(λ*x 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 add*: *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*) (λT_f. *if* $P \vdash T \leq$ *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*) (λT_f. *if* $P \vdash T_2 \leq$ (*Class C*) ∧ $P \vdash T_1 \leq T_f$ *then Predicate*.*single* () *else bot*))
by(*auto simp add*: *Predicate*.*holds-eq simp del*: *eval-bind split*: *split-if-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 split-if-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*-prog* = $expr_1$ *prog*
type-synonym $state_1$ = *heap* × (*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*-prog* ⇒ $expr_1$ ⇒ $state_1$ ⇒ $expr_1$ ⇒ $state_1$ ⇒ *bool*
      (- $\vdash_1$ ((*1*⟨-,/-⟩) ⇒/ (*1*⟨-,/-⟩)) [*51*,*0*,*0*,*0*,*0*] *81*)
  and $evals_1$ :: J_1*-prog* ⇒ $expr_1$ *list* ⇒ $state_1$ ⇒ $expr_1$ *list* ⇒ $state_1$ ⇒ *bool*
      (- $\vdash_1$ ((*1*⟨-,/-⟩) [⇒]/ (*1*⟨-,/-⟩)) [*51*,*0*,*0*,*0*,*0*] *81*)
  for *P* :: J_1*-prog*
where

  New_1:
  ⟦ *new-Addr h* = *Some a*; *P* ⊢ *C has-fields FDTs*; *h'* = *h*(*a*↦(*C*,*init-fields FDTs*)) ⟧
  ⟹ *P* $\vdash_1$ ⟨*new C*,(*h*,*l*)⟩ ⇒ ⟨*addr a*,(*h'*,*l*)⟩
| $NewFail_1$:
  new-Addr h = *None* ⟹
  P $\vdash_1$ ⟨*new C*, (*h*,*l*)⟩ ⇒ ⟨*THROW OutOfMemory*,(*h*,*l*)⟩

| $Cast_1$:
  ⟦ *P* $\vdash_1$ ⟨*e*,s_0⟩ ⇒ ⟨*addr a*,(*h*,*l*)⟩; *h a* = *Some*(*D*,*fs*); *P* ⊢ *D* $\preceq^*$ *C* ⟧
  ⟹ *P* $\vdash_1$ ⟨*Cast C e*,s_0⟩ ⇒ ⟨*addr a*,(*h*,*l*)⟩
| $CastNull_1$:
  P $\vdash_1$ ⟨*e*,s_0⟩ ⇒ ⟨*null*,s_1⟩ ⟹
  P $\vdash_1$ ⟨*Cast C e*,s_0⟩ ⇒ ⟨*null*,s_1⟩
| $CastFail_1$:

$\llbracket\ P \vdash_1 \langle e,s_0\rangle \Rightarrow \langle \mathit{addr}\ a,(h,l)\rangle;\ h\ a = \mathit{Some}(D,\mathit{fs});\ \neg\ P \vdash D \preceq^* C\ \rrbracket$
$\Longrightarrow P \vdash_1 \langle \mathit{Cast}\ C\ e,s_0\rangle \Rightarrow \langle \mathit{THROW}\ \mathit{ClassCast},(h,l)\rangle$
| $\mathit{CastThrow}_1$:
$P \vdash_1 \langle e,s_0\rangle \Rightarrow \langle \mathit{throw}\ e',s_1\rangle \Longrightarrow$
$P \vdash_1 \langle \mathit{Cast}\ C\ e,s_0\rangle \Rightarrow \langle \mathit{throw}\ e',s_1\rangle$

| Val_1:
$P \vdash_1 \langle \mathit{Val}\ v,s\rangle \Rightarrow \langle \mathit{Val}\ v,s\rangle$

| BinOp_1:
$\llbracket\ P \vdash_1 \langle e_1,s_0\rangle \Rightarrow \langle \mathit{Val}\ v_1,s_1\rangle;\ P \vdash_1 \langle e_2,s_1\rangle \Rightarrow \langle \mathit{Val}\ v_2,s_2\rangle;\ \mathit{binop}(\mathit{bop},v_1,v_2) = \mathit{Some}\ v\ \rrbracket$
$\Longrightarrow P \vdash_1 \langle e_1 \ll\mathit{bop}\gg e_2,s_0\rangle \Rightarrow \langle \mathit{Val}\ v,s_2\rangle$
| $\mathit{BinOpThrow}_{11}$:
$P \vdash_1 \langle e_1,s_0\rangle \Rightarrow \langle \mathit{throw}\ e,s_1\rangle \Longrightarrow$
$P \vdash_1 \langle e_1 \ll\mathit{bop}\gg e_2,\ s_0\rangle \Rightarrow \langle \mathit{throw}\ e,s_1\rangle$
| $\mathit{BinOpThrow}_{21}$:
$\llbracket\ P \vdash_1 \langle e_1,s_0\rangle \Rightarrow \langle \mathit{Val}\ v_1,s_1\rangle;\ P \vdash_1 \langle e_2,s_1\rangle \Rightarrow \langle \mathit{throw}\ e,s_2\rangle\ \rrbracket$
$\Longrightarrow P \vdash_1 \langle e_1 \ll\mathit{bop}\gg e_2,s_0\rangle \Rightarrow \langle \mathit{throw}\ e,s_2\rangle$

| Var_1:
$\llbracket\ \mathit{ls}!i = v;\ i < \mathit{size}\ \mathit{ls}\ \rrbracket \Longrightarrow$
$P \vdash_1 \langle \mathit{Var}\ i,(h,\mathit{ls})\rangle \Rightarrow \langle \mathit{Val}\ v,(h,\mathit{ls})\rangle$

| LAss_1:
$\llbracket\ P \vdash_1 \langle e,s_0\rangle \Rightarrow \langle \mathit{Val}\ v,(h,\mathit{ls})\rangle;\ i < \mathit{size}\ \mathit{ls};\ \mathit{ls}' = \mathit{ls}[i := v]\ \rrbracket$
$\Longrightarrow P \vdash_1 \langle i{:=}\ e,s_0\rangle \Rightarrow \langle \mathit{unit},(h,\mathit{ls}')\rangle$
| $\mathit{LAssThrow}_1$:
$P \vdash_1 \langle e,s_0\rangle \Rightarrow \langle \mathit{throw}\ e',s_1\rangle \Longrightarrow$
$P \vdash_1 \langle i{:=}\ e,s_0\rangle \Rightarrow \langle \mathit{throw}\ e',s_1\rangle$

| FAcc_1:
$\llbracket\ P \vdash_1 \langle e,s_0\rangle \Rightarrow \langle \mathit{addr}\ a,(h,\mathit{ls})\rangle;\ h\ a = \mathit{Some}(C,\mathit{fs});\ \mathit{fs}(F,D) = \mathit{Some}\ v\ \rrbracket$
$\Longrightarrow P \vdash_1 \langle e{\cdot}F\{D\},s_0\rangle \Rightarrow \langle \mathit{Val}\ v,(h,\mathit{ls})\rangle$
| $\mathit{FAccNull}_1$:
$P \vdash_1 \langle e,s_0\rangle \Rightarrow \langle \mathit{null},s_1\rangle \Longrightarrow$
$P \vdash_1 \langle e{\cdot}F\{D\},s_0\rangle \Rightarrow \langle \mathit{THROW}\ \mathit{NullPointer},s_1\rangle$
| $\mathit{FAccThrow}_1$:
$P \vdash_1 \langle e,s_0\rangle \Rightarrow \langle \mathit{throw}\ e',s_1\rangle \Longrightarrow$
$P \vdash_1 \langle e{\cdot}F\{D\},s_0\rangle \Rightarrow \langle \mathit{throw}\ e',s_1\rangle$

| FAss_1:
$\llbracket\ P \vdash_1 \langle e_1,s_0\rangle \Rightarrow \langle \mathit{addr}\ a,s_1\rangle;\ P \vdash_1 \langle e_2,s_1\rangle \Rightarrow \langle \mathit{Val}\ v,(h_2,l_2)\rangle;$
$h_2\ a = \mathit{Some}(C,\mathit{fs});\ \mathit{fs}' = \mathit{fs}((F,D)\mapsto v);\ h_2' = h_2(a\mapsto(C,\mathit{fs}'))\ \rrbracket$
$\Longrightarrow P \vdash_1 \langle e_1{\cdot}F\{D\}{:=}\ e_2,s_0\rangle \Rightarrow \langle \mathit{unit},(h_2',l_2)\rangle$
| $\mathit{FAssNull}_1$:
$\llbracket\ P \vdash_1 \langle e_1,s_0\rangle \Rightarrow \langle \mathit{null},s_1\rangle;\ \ P \vdash_1 \langle e_2,s_1\rangle \Rightarrow \langle \mathit{Val}\ v,s_2\rangle\ \rrbracket$
$\Longrightarrow P \vdash_1 \langle e_1{\cdot}F\{D\}{:=}\ e_2,s_0\rangle \Rightarrow \langle \mathit{THROW}\ \mathit{NullPointer},s_2\rangle$
| $\mathit{FAssThrow}_{11}$:
$P \vdash_1 \langle e_1,s_0\rangle \Rightarrow \langle \mathit{throw}\ e',s_1\rangle \Longrightarrow$
$P \vdash_1 \langle e_1{\cdot}F\{D\}{:=}\ e_2,s_0\rangle \Rightarrow \langle \mathit{throw}\ e',s_1\rangle$
| $\mathit{FAssThrow}_{21}$:
$\llbracket\ P \vdash_1 \langle e_1,s_0\rangle \Rightarrow \langle \mathit{Val}\ v,s_1\rangle;\ P \vdash_1 \langle e_2,s_1\rangle \Rightarrow \langle \mathit{throw}\ e',s_2\rangle\ \rrbracket$
$\Longrightarrow P \vdash_1 \langle e_1{\cdot}F\{D\}{:=}\ e_2,s_0\rangle \Rightarrow \langle \mathit{throw}\ e',s_2\rangle$

| *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*$_1$:
$\llbracket\ P \vdash_1 \langle e,s_0\rangle \Rightarrow \langle null,s_1\rangle;\ P \vdash_1 \langle es,s_1\rangle\ [\Rightarrow]\ \langle map\ Val\ vs,s_2\rangle\ \rrbracket$
$\Longrightarrow P \vdash_1 \langle e{\cdot}M(es),s_0\rangle \Rightarrow \langle THROW\ NullPointer,s_2\rangle$
| *Call*$_1$:
$\llbracket\ P \vdash_1 \langle e,s_0\rangle \Rightarrow \langle addr\ a,s_1\rangle;\ P \vdash_1 \langle es,s_1\rangle\ [\Rightarrow]\ \langle map\ Val\ vs,(h_2,ls_2)\rangle;$
$h_2\ a = Some(C,fs);\ P \vdash C\ sees\ M{:}Ts{\rightarrow}T = body\ in\ D;$
$size\ vs = size\ Ts;\ ls_2{}' = (Addr\ a)\ \#\ vs\ @\ replicate\ (max\text{-}vars\ body)\ undefined;$
$P \vdash_1 \langle body,(h_2,ls_2{}')\rangle \Rightarrow \langle e',(h_3,ls_3)\rangle\ \rrbracket$
$\Longrightarrow P \vdash_1 \langle e{\cdot}M(es),s_0\rangle \Rightarrow \langle e',(h_3,ls_2)\rangle$
| *CallParamsThrow*$_1$:
$\llbracket\ P \vdash_1 \langle e,s_0\rangle \Rightarrow \langle Val\ v,s_1\rangle;\ P \vdash_1 \langle es,s_1\rangle\ [\Rightarrow]\ \langle es',s_2\rangle;$
$es' = map\ Val\ vs\ @\ throw\ ex\ \#\ es_2\ \rrbracket$
$\Longrightarrow P \vdash_1 \langle e{\cdot}M(es),s_0\rangle \Rightarrow \langle throw\ ex,s_2\rangle$

| *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*$_1$:
$\llbracket\ P \vdash_1 \langle e_0,s_0\rangle \Rightarrow \langle Val\ v,s_1\rangle;\ P \vdash_1 \langle e_1,s_1\rangle \Rightarrow \langle e_2,s_2\rangle\ \rrbracket$
$\Longrightarrow P \vdash_1 \langle e_0;;e_1,s_0\rangle \Rightarrow \langle e_2,s_2\rangle$
| *SeqThrow*$_1$:
$P \vdash_1 \langle e_0,s_0\rangle \Rightarrow \langle throw\ e,s_1\rangle \Longrightarrow$
$P \vdash_1 \langle e_0;;e_1,s_0\rangle \Rightarrow \langle throw\ e,s_1\rangle$

| *CondT*$_1$:
$\llbracket\ P \vdash_1 \langle e,s_0\rangle \Rightarrow \langle true,s_1\rangle;\ P \vdash_1 \langle e_1,s_1\rangle \Rightarrow \langle e',s_2\rangle\ \rrbracket$
$\Longrightarrow P \vdash_1 \langle if\ (e)\ e_1\ else\ e_2,s_0\rangle \Rightarrow \langle e',s_2\rangle$
| *CondF*$_1$:
$\llbracket\ P \vdash_1 \langle e,s_0\rangle \Rightarrow \langle false,s_1\rangle;\ P \vdash_1 \langle e_2,s_1\rangle \Rightarrow \langle e',s_2\rangle\ \rrbracket$
$\Longrightarrow P \vdash_1 \langle if\ (e)\ e_1\ else\ e_2,s_0\rangle \Rightarrow \langle e',s_2\rangle$
| *CondThrow*$_1$:
$P \vdash_1 \langle e,s_0\rangle \Rightarrow \langle throw\ e',s_1\rangle \Longrightarrow$
$P \vdash_1 \langle if\ (e)\ e_1\ else\ e_2,\ s_0\rangle \Rightarrow \langle throw\ e',s_1\rangle$

| *WhileF*$_1$:
$P \vdash_1 \langle e,s_0\rangle \Rightarrow \langle false,s_1\rangle \Longrightarrow$
$P \vdash_1 \langle while\ (e)\ c,s_0\rangle \Rightarrow \langle unit,s_1\rangle$
| *WhileT*$_1$:
$\llbracket\ P \vdash_1 \langle e,s_0\rangle \Rightarrow \langle true,s_1\rangle;\ P \vdash_1 \langle c,s_1\rangle \Rightarrow \langle Val\ v_1,s_2\rangle;$
$P \vdash_1 \langle while\ (e)\ c,s_2\rangle \Rightarrow \langle e_3,s_3\rangle\ \rrbracket$
$\Longrightarrow P \vdash_1 \langle while\ (e)\ c,s_0\rangle \Rightarrow \langle e_3,s_3\rangle$
| *WhileCondThrow*$_1$:
$P \vdash_1 \langle e,s_0\rangle \Rightarrow \langle throw\ e',s_1\rangle \Longrightarrow$
$P \vdash_1 \langle while\ (e)\ c,s_0\rangle \Rightarrow \langle throw\ e',s_1\rangle$
| *WhileBodyThrow*$_1$:
$\llbracket\ P \vdash_1 \langle e,s_0\rangle \Rightarrow \langle true,s_1\rangle;\ P \vdash_1 \langle c,s_1\rangle \Rightarrow \langle throw\ e',s_2\rangle\rrbracket$
$\Longrightarrow P \vdash_1 \langle while\ (e)\ c,s_0\rangle \Rightarrow \langle throw\ e',s_2\rangle$

| *Throw*$_1$:
$P \vdash_1 \langle e,s_0\rangle \Rightarrow \langle addr\ a,s_1\rangle \Longrightarrow$
$P \vdash_1 \langle throw\ e,s_0\rangle \Rightarrow \langle Throw\ a,s_1\rangle$

| *ThrowNull*$_1$:
$P \vdash_1 \langle e,s_0\rangle \Rightarrow \langle null,s_1\rangle \Longrightarrow$
$P \vdash_1 \langle throw\ e,s_0\rangle \Rightarrow \langle THROW\ NullPointer,s_1\rangle$
| *ThrowThrow*$_1$:
$P \vdash_1 \langle e,s_0\rangle \Rightarrow \langle throw\ e',s_1\rangle \Longrightarrow$
$P \vdash_1 \langle throw\ e,s_0\rangle \Rightarrow \langle throw\ e',s_1\rangle$

| *Try*$_1$:
$P \vdash_1 \langle e_1,s_0\rangle \Rightarrow \langle Val\ v_1,s_1\rangle \Longrightarrow$
$P \vdash_1 \langle try\ e_1\ catch(C\ i)\ e_2,s_0\rangle \Rightarrow \langle Val\ v_1,s_1\rangle$
| *TryCatch*$_1$:
$[\![\ P \vdash_1 \langle e_1,s_0\rangle \Rightarrow \langle Throw\ a,(h_1,ls_1)\rangle;$
$h_1\ a = Some(D,fs);\ P \vdash D \preceq^* C;\ i < length\ ls_1;$
$P \vdash_1 \langle e_2,(h_1,ls_1[i:=Addr\ a])\rangle \Rightarrow \langle e_2',(h_2,ls_2)\rangle\ ]\!]$
$\Longrightarrow P \vdash_1 \langle try\ e_1\ catch(C\ i)\ e_2,s_0\rangle \Rightarrow \langle e_2',(h_2,ls_2)\rangle$
| *TryThrow*$_1$:
$[\![\ P \vdash_1 \langle e_1,s_0\rangle \Rightarrow \langle Throw\ a,(h_1,ls_1)\rangle;\ h_1\ a = Some(D,fs);\ \neg\ P \vdash D \preceq^* C\ ]\!]$
$\Longrightarrow P \vdash_1 \langle try\ e_1\ catch(C\ i)\ e_2,s_0\rangle \Rightarrow \langle Throw\ a,(h_1,ls_1)\rangle$

| *Nil*$_1$:
$P \vdash_1 \langle [],s\rangle\ [\Rightarrow]\ \langle [],s\rangle$

| *Cons*$_1$:
$[\![\ P \vdash_1 \langle e,s_0\rangle \Rightarrow \langle Val\ v,s_1\rangle;\ P \vdash_1 \langle es,s_1\rangle\ [\Rightarrow]\ \langle es',s_2\rangle\ ]\!]$
$\Longrightarrow P \vdash_1 \langle e\#es,s_0\rangle\ [\Rightarrow]\ \langle Val\ v\ \#\ es',s_2\rangle$
| *ConsThrow*$_1$:
$P \vdash_1 \langle e,s_0\rangle \Rightarrow \langle throw\ e',s_1\rangle \Longrightarrow$
$P \vdash_1 \langle e\#es,s_0\rangle\ [\Rightarrow]\ \langle throw\ e'\ \#\ es,\ s_1\rangle$

lemma *eval*$_1$*-preserves-len*:
$P \vdash_1 \langle e_0,(h_0,ls_0)\rangle \Rightarrow \langle e_1,(h_1,ls_1)\rangle \Longrightarrow length\ ls_0 = length\ ls_1$
and *evals*$_1$*-preserves-len*:
$P \vdash_1 \langle es_0,(h_0,ls_0)\rangle\ [\Rightarrow]\ \langle es_1,(h_1,ls_1)\rangle \Longrightarrow length\ ls_0 = length\ ls_1$

lemma *evals*$_1$*-preserves-elen*:
$\bigwedge es'\ s\ s'.\ P \vdash_1 \langle es,s\rangle\ [\Rightarrow]\ \langle es',s'\rangle \Longrightarrow length\ es = length\ es'$

lemma *eval*$_1$*-final*: $P \vdash_1 \langle e,s\rangle \Rightarrow \langle e',s'\rangle \Longrightarrow final\ e'$
and *evals*$_1$*-final*: $P \vdash_1 \langle es,s\rangle\ [\Rightarrow]\ \langle es',s'\rangle \Longrightarrow finals\ es'$

end

5.2 Well-Formedness of Intermediate Language

theory *J1WellForm*
imports *../J/JWellForm J1*
begin

5.2.1 Well-Typedness

type-synonym
 env_1 = *ty list* — type environment indexed by variable number

inductive
 $WT_1 :: [J_1\text{-}prog, env_1, expr_1, ty] \Rightarrow bool$
  $((\text{-},\text{-} \vdash_1/ \text{-} :: \text{-})\ [51,51,51]50)$
 and $WTs_1 :: [J_1\text{-}prog, env_1, expr_1\ list, ty\ list] \Rightarrow bool$
  $((\text{-},\text{-} \vdash_1/ \text{-} [::] \text{-})\ [51,51,51]50)$
 for $P :: J_1\text{-}prog$
where

$WTNew_1$:
$is\text{-}class\ P\ C \Longrightarrow$
$P,E \vdash_1 new\ C :: Class\ C$

| $WTCast_1$:
$[\![\ P,E \vdash_1 e :: Class\ D;\ is\text{-}class\ P\ C;\ P \vdash C \preceq^* D \vee P \vdash D \preceq^* C\ ]\!]$
$\Longrightarrow P,E \vdash_1 Cast\ C\ e :: Class\ C$

| $WTVal_1$:
$typeof\ v = Some\ T \Longrightarrow$
$P,E \vdash_1 Val\ v :: T$

| $WTVar_1$:
$[\![\ E!i = T;\ i < size\ E\ ]\!]$
$\Longrightarrow P,E \vdash_1 Var\ i :: T$

| $WTBinOp_1$:
$[\![\ P,E \vdash_1 e_1 :: T_1;\ P,E \vdash_1 e_2 :: T_2;$
 $case\ bop\ of\ Eq \Rightarrow (P \vdash T_1 \leq T_2 \vee P \vdash T_2 \leq T_1) \wedge T = Boolean$
  $|\ Add \Rightarrow T_1 = Integer \wedge T_2 = Integer \wedge T = Integer\ ]\!]$
$\Longrightarrow P,E \vdash_1 e_1 \ll bop \gg e_2 :: T$

| $WTLAss_1$:
$[\![\ E!i = T;\ i < size\ E;\ P,E \vdash_1 e :: T';\ P \vdash T' \leq T\ ]\!]$
$\Longrightarrow P,E \vdash_1 i{:=}e :: Void$

| $WTFAcc_1$:
$[\![\ P,E \vdash_1 e :: Class\ C;\ P \vdash C\ sees\ F{:}T\ in\ D\ ]\!]$
$\Longrightarrow P,E \vdash_1 e{\cdot}F\{D\} :: T$

| $WTFAss_1$:
$[\![\ P,E \vdash_1 e_1 :: Class\ C;\ P \vdash C\ sees\ F{:}T\ in\ D;\ P,E \vdash_1 e_2 :: T';\ P \vdash T' \leq T\ ]\!]$
$\Longrightarrow P,E \vdash_1 e_1{\cdot}F\{D\} := e_2 :: Void$

| $WTCall_1$:

$[\![$ $P,E \vdash_1 e :: Class\ C$; $P \vdash C\ sees\ M{:}Ts' \rightarrow T = m\ in\ D$;
$P,E \vdash_1 es\ [::]\ Ts$; $P \vdash Ts\ [\leq]\ Ts'$ $]\!]$
$\Longrightarrow P,E \vdash_1 e{\bullet}M(es) :: T$

| $WTBlock_1$:
$[\![$ $is\text{-}type\ P\ T$; $P,E@[T] \vdash_1 e{::}T'$ $]\!]$
$\Longrightarrow$ $P,E \vdash_1 \{i{:}T;\ e\} :: T'$

| $WTSeq_1$:
$[\![$ $P,E \vdash_1 e_1{::}T_1$; $P,E \vdash_1 e_2{::}T_2$ $]\!]$
$\Longrightarrow$ $P,E \vdash_1 e_1;;e_2 :: T_2$

| $WTCond_1$:
$[\![$ $P,E \vdash_1 e :: Boolean$; $P,E \vdash_1 e_1{::}T_1$; $P,E \vdash_1 e_2{::}T_2$;
$P \vdash T_1 \leq T_2 \vee P \vdash T_2 \leq T_1$; $P \vdash T_1 \leq T_2 \longrightarrow T = T_2$; $P \vdash T_2 \leq T_1 \longrightarrow T = T_1$ $]\!]$
$\Longrightarrow P,E \vdash_1 if\ (e)\ e_1\ else\ e_2 :: T$

| $WTWhile_1$:
$[\![$ $P,E \vdash_1 e :: Boolean$; $P,E \vdash_1 c{::}T$ $]\!]$
$\Longrightarrow P,E \vdash_1 while\ (e)\ c :: Void$

| $WTThrow_1$:
$P,E \vdash_1 e :: Class\ C$ $\Longrightarrow$
$P,E \vdash_1 throw\ e :: Void$

| $WTTry_1$:
$[\![$ $P,E \vdash_1 e_1 :: T$; $P,E@[Class\ C] \vdash_1 e_2 :: T$; $is\text{-}class\ P\ C$ $]\!]$
$\Longrightarrow P,E \vdash_1 try\ e_1\ catch(C\ i)\ e_2 :: T$

| $WTNil_1$:
$P,E \vdash_1 []\ [::]\ []$

| $WTCons_1$:
$[\![$ $P,E \vdash_1 e :: T$; $P,E \vdash_1 es\ [::]\ Ts$ $]\!]$
$\Longrightarrow$ $P,E \vdash_1 e\#es\ [::]\ T\#Ts$

lemma $WTs_1\text{-}same\text{-}size$: $\bigwedge Ts.\ P,E \vdash_1 es\ [::]\ Ts \Longrightarrow size\ es = size\ Ts$

lemma $WT_1\text{-}unique$:
$P,E \vdash_1 e :: T_1 \Longrightarrow (\bigwedge T_2.\ P,E \vdash_1 e :: T_2 \Longrightarrow T_1 = T_2)$ **and**
$P,E \vdash_1 es\ [::]\ Ts_1 \Longrightarrow (\bigwedge Ts_2.\ P,E \vdash_1 es\ [::]\ Ts_2 \Longrightarrow Ts_1 = Ts_2)$

lemma assumes *wf*: *wf-prog p P*
shows $WT_1\text{-}is\text{-}type$: $P,E \vdash_1 e :: T \Longrightarrow set\ E \subseteq types\ P \Longrightarrow is\text{-}type\ P\ T$
and $P,E \vdash_1 es\ [::]\ Ts \Longrightarrow True$

5.2.2 Well-formedness

— Indices in blocks increase by 1

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* ∧ $\mathcal{B}$ e_2 *i*) |
$\mathcal{B}$ (*Var j*) *i* = *True* |
$\mathcal{B}$ (*e*•*F*{*D*}) *i* = $\mathcal{B}$ *e i* |
$\mathcal{B}$ (*j*:=*e*) *i* = $\mathcal{B}$ *e i* |
$\mathcal{B}$ (e_1•*F*{*D*} := e_2) *i* = ($\mathcal{B}$ e_1 *i* ∧ $\mathcal{B}$ e_2 *i*) |
$\mathcal{B}$ (*e*•*M*(*es*)) *i* = ($\mathcal{B}$ *e i* ∧ $\mathcal{B}$*s es i*) |
$\mathcal{B}$ ({*j*:*T* ; *e*}) *i* = (*i* = *j* ∧ $\mathcal{B}$ *e* (*i*+*1*)) |
$\mathcal{B}$ (e_1;;e_2) *i* = ($\mathcal{B}$ e_1 *i* ∧ $\mathcal{B}$ e_2 *i*) |
$\mathcal{B}$ (*if* (*e*) e_1 *else* e_2) *i* = ($\mathcal{B}$ *e i* ∧ $\mathcal{B}$ e_1 *i* ∧ $\mathcal{B}$ e_2 *i*) |
$\mathcal{B}$ (*throw e*) *i* = $\mathcal{B}$ *e i* |
$\mathcal{B}$ (*while* (*e*) *c*) *i* = ($\mathcal{B}$ *e i* ∧ $\mathcal{B}$ *c i*) |
$\mathcal{B}$ (*try* e_1 *catch*(*C j*) e_2) *i* = ($\mathcal{B}$ e_1 *i* ∧ *i*=*j* ∧ $\mathcal{B}$ e_2 (*i*+*1*)) |

$\mathcal{B}$*s* [] *i* = *True* |
$\mathcal{B}$*s* (*e*#*es*) *i* = ($\mathcal{B}$ *e i* ∧ $\mathcal{B}$*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* $\vdash_1$ *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* $\vdash_1$ *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* :: $('a \Rightarrow 'b) \Rightarrow 'a\ mdecl \Rightarrow 'b\ mdecl$
where
$compM\ f \equiv \lambda(M,\ Ts,\ T,\ m).\ (M,\ Ts,\ T,\ f\ m)$

definition *compC* :: $('a \Rightarrow 'b) \Rightarrow 'a\ cdecl \Rightarrow 'b\ cdecl$
where
$compC\ f \equiv \lambda(C,D,Fdecls,Mdecls).\ (C,D,Fdecls,\ map\ (compM\ f)\ Mdecls)$

definition *compP* :: $('a \Rightarrow 'b) \Rightarrow 'a\ prog \Rightarrow 'b\ prog$
where
$compP\ f \equiv map\ (compC\ f)$

Compilation preserves the program structure. Therfore lookup functions either commute with compilation (like method lookup) or are preserved by it (like the subclass relation).

lemma *map-of-map4*:
$map\text{-}of\ (map\ (\lambda(x,a,b,c).(x,a,b,f\ c))\ ts) =$
$Option.map\ (\lambda(a,b,c).(a,b,f\ c)) \circ (map\text{-}of\ ts)$

lemma *class-compP*:
$class\ P\ C = Some\ (D,\ fs,\ ms)$
$\Longrightarrow class\ (compP\ f\ P)\ C = Some\ (D,\ fs,\ map\ (compM\ f)\ ms)$

lemma *class-compPD*:
$class\ (compP\ f\ P)\ C = Some\ (D,\ fs,\ cms)$
$\Longrightarrow \exists ms.\ class\ P\ C = Some(D,fs,ms) \wedge cms = map\ (compM\ f)\ ms$

lemma [*simp*]: $is\text{-}class\ (compP\ f\ P)\ C = is\text{-}class\ P\ C$

lemma [*simp*]: $class\ (compP\ f\ P)\ C = Option.map\ (\lambda c.\ snd(compC\ f\ (C,c)))\ (class\ P\ C)$

lemma *sees-methods-compP*:
$P \vdash C\ sees\text{-}methods\ Mm \Longrightarrow$
$compP\ f\ P \vdash C\ sees\text{-}methods\ (Option.map\ (\lambda((Ts,T,m),D).\ ((Ts,T,f\ m),D)) \circ Mm)$

lemma *sees-method-compP*:
$P \vdash C\ sees\ M:\ Ts \rightarrow T = m\ in\ D \Longrightarrow$
$compP\ f\ P \vdash C\ sees\ M:\ Ts \rightarrow T = (f\ m)\ in\ D$

lemma [*simp*]:
$P \vdash C\ sees\ M:\ Ts \rightarrow T = m\ in\ D \Longrightarrow$
$method\ (compP\ f\ P)\ C\ M = (D,Ts,T,f\ m)$

lemma *sees-methods-compPD*:
$[\![\ cP \vdash C\ sees\text{-}methods\ Mm';\ cP = compP\ f\ P\ ]\!] \Longrightarrow$
$\exists Mm.\ P \vdash C\ sees\text{-}methods\ Mm \wedge$
$\quad Mm' = (Option.map\ (\lambda((Ts,T,m),D).\ ((Ts,T,f\ m),D)) \circ Mm)$

lemma *sees-method-compPD*:

compP f P $\vdash$ *C sees M*: $Ts \to T$ = *fm in D* $\Longrightarrow$
$\exists$ *m*. *P* $\vdash$ *C sees M*: $Ts \to T$ = *m in D* $\wedge$ *f m* = *fm*

lemma [*simp*]: *subcls1*(*compP f P*) = *subcls1 P*

lemma *compP-widen*[*simp*]: (*compP f P* $\vdash$ $T \leq T'$) = (*P* $\vdash$ $T \leq T'$)

lemma [*simp*]: (*compP f P* $\vdash$ *Ts* [$\leq$] *Ts'*) = (*P* $\vdash$ *Ts* [$\leq$] *Ts'*)

lemma [*simp*]: *is-type* (*compP f P*) *T* = *is-type P T*

lemma [*simp*]: (*compP* (*f*::'*a*$\Rightarrow$'*b*) *P* $\vdash$ *C has-fields FDTs*) = (*P* $\vdash$ *C has-fields FDTs*)

lemma [*simp*]: *fields* (*compP f P*) *C* = *fields P C*

lemma [*simp*]: (*compP f P* $\vdash$ *C sees F*:*T in D*) = (*P* $\vdash$ *C sees F*:*T in D*)

lemma [*simp*]: *field* (*compP f P*) *F D* = *field P F D*

5.3.1 Invariance of *wf-prog* under compilation

lemma [*iff*]: *distinct-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'*) $\in$ *set*(*compP f P*)) =
($\exists$ *ms*. (*C*,*D*,*fs*,*ms*) $\in$ *set P* $\wedge$ *ms'* = *map* (*compM f*) *ms*)

lemma *wf-cdecl-compPI*:
$[\![$ $\bigwedge$*C M Ts T m*.
$[\![$ *wf-mdecl* wf_1 *P C* (*M*,*Ts*,*T*,*m*); *P* $\vdash$ *C sees M*:$Ts \to T$ = *m in C* $]\!]$
$\Longrightarrow$ *wf-mdecl* wf_2 (*compP f P*) *C* (*M*,*Ts*,*T*, *f m*);
$\forall$ *x*$\in$*set P*. *wf-cdecl* wf_1 *P x*; *x* $\in$ *set* (*compP f P*); *wf-prog p P* $]\!]$
$\Longrightarrow$ *wf-cdecl* wf_2 (*compP f P*) *x*

lemma *wf-prog-compPI*:
assumes *lift*:
$\bigwedge$*C M Ts T m*.
$[\![$ *P* $\vdash$ *C sees M*:$Ts \to T$ = *m in C*; *wf-mdecl* wf_1 *P C* (*M*,*Ts*,*T*,*m*) $]\!]$
$\Longrightarrow$ *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 *List-Index* **imports** *Main* **begin**

This theory defines three functions for finding the index of items in a list:

find-index P xs finds the index of the first element in *xs* that satisfies *P*.

index xs x finds the index of the first occurrence of *x* in *xs*.

last-index xs x finds the index of the last occurrence of *x* in *xs*.

All functions return *length xs* if *xs* does not contain a suitable element.

The argument order of *find-index* follows the function of the same name in the Haskell standard library. For *index* (and *last-index*) the order is intentionally reversed: *index* maps lists to a mapping from elements to their indices, almost the inverse of function *nth*.

primrec *find-index* :: $('a \Rightarrow bool) \Rightarrow 'a\ list \Rightarrow nat$ **where**
find-index - [] = *0* |
find-index P (*x#xs*) = (*if P x then 0 else find-index P xs* + *1*)

definition *index* :: $'a\ list \Rightarrow 'a \Rightarrow nat$ **where**
index xs = (λa. *find-index* (λx. *x*=*a*) *xs*)

definition *last-index* :: $'a\ list \Rightarrow 'a \Rightarrow nat$ **where**
last-index xs x =
(*let i* = *index* (*rev xs*) *x*; *n* = *size xs*
in if i = *n then i else n* − (*i*+*1*))

lemma *find-index-le-size*: *find-index P xs* <= *size xs*
by(*induct xs*) *simp-all*

lemma *index-le-size*: *index xs x* <= *size xs*
by(*simp add*: *index-def find-index-le-size*)

lemma *last-index-le-size*: *last-index xs x* <= *size xs*
by(*simp add*: *last-index-def Let-def index-le-size*)

lemma *index-Nil*[*simp*]: *index* [] *a* = *0*
by(*simp add*: *index-def*)

lemma *index-Cons*[*simp*]: *index* (*x#xs*) *a* = (*if x*=*a then 0 else index xs a* + *1*)
by(*simp add*: *index-def*)

lemma *index-append*: *index* (*xs* @ *ys*) *x* =
(*if x* : *set xs then index xs x else size xs* + *index ys x*)
by (*induct xs*) *simp-all*

lemma *index-conv-size-if-notin*[*simp*]: $x \notin set\ xs \Longrightarrow index\ xs\ x = size\ xs$
by (*induct xs*) *auto*

lemma *find-index-eq-size-conv*:
$size\ xs = n \Longrightarrow (find\text{-}index\ P\ xs = n) = (ALL\ x : set\ xs.\ {\sim}\ P\ x)$
by(*induct xs arbitrary*: *n*) *auto*

lemma *size-eq-find-index-conv*:
$size\ xs = n \Longrightarrow (n = find\text{-}index\ P\ xs) = (ALL\ x : set\ xs.\ {\sim}\ P\ x)$
by(*metis find-index-eq-size-conv*)

lemma *index-size-conv*: $size\ xs = n \Longrightarrow (index\ xs\ x = n) = (x \notin set\ xs)$
by(*auto simp*: *index-def find-index-eq-size-conv*)

lemma *size-index-conv*: *size xs* = *n* ⟹ (*n* = *index xs x*) = (*x* ∉ *set xs*)
by (*metis index-size-conv*)

lemma *last-index-size-conv*:
size xs = *n* ⟹ (*last-index xs x* = *n*) = (*x* ∉ *set xs*)
apply(*auto simp*: *last-index-def index-size-conv*)
apply(*drule length-pos-if-in-set*)
apply *arith*
done

lemma *size-last-index-conv*:
size xs = *n* ⟹ (*n* = *last-index xs x*) = (*x* ∉ *set xs*)
by (*metis last-index-size-conv*)

lemma *find-index-less-size-conv*:
(*find-index P xs* < *size xs*) = (*EX x* : *set xs*. *P x*)
by (*induct xs*) *auto*

lemma *index-less-size-conv*:
(*index xs x* < *size xs*) = (*x* ∈ *set xs*)
by(*auto simp*: *index-def find-index-less-size-conv*)

lemma *last-index-less-size-conv*:
(*last-index xs x* < *size xs*) = (*x* : *set xs*)
by(*simp add*: *last-index-def Let-def index-size-conv length-pos-if-in-set*
del:*length-greater-0-conv*)

lemma *index-less*[*simp*]:
x : *set xs* ⟹ *size xs* <= *n* ⟹ *index xs x* < *n*
apply(*induct xs*) **apply** *auto*
apply (*metis index-less-size-conv less-eq-Suc-le less-trans-Suc*)
done

lemma *last-index-less*[*simp*]:
x : *set xs* ⟹ *size xs* <= *n* ⟹ *last-index xs x* < *n*
by(*simp add*: *last-index-less-size-conv*[*symmetric*])

lemma *last-index-Cons*: *last-index* (*x*#*xs*) *y* =
(*if x*=*y then*
if x ∈ *set xs then last-index xs y* + *1 else 0*
else last-index xs y + *1*)
using *index-le-size*[*of rev xs y*]
apply(*auto simp add*: *last-index-def index-append Let-def*)
apply(*simp add*: *index-size-conv*)
done

lemma *last-index-append*: *last-index* (*xs* @ *ys*) *x* =
(*if x* : *set ys then size xs* + *last-index ys x*
else if x : *set xs then last-index xs x else size xs* + *size ys*)
by (*induct xs*) (*simp-all add*: *last-index-Cons last-index-size-conv*)

lemma *last-index-Snoc*[*simp*]:
last-index (*xs* @ [*x*]) *y* =

```
  (if x=y then size xs
   else if y : set xs then last-index xs y else size xs + 1)
by(simp add: last-index-append last-index-Cons)

lemma nth-find-index: find-index P xs < size xs ⟹ P(xs ! find-index P xs)
by (induct xs) auto

lemma nth-index[simp]: x ∈ set xs ⟹ xs ! index xs x = x
by (induct xs) auto

lemma nth-last-index[simp]: x ∈ set xs ⟹ xs ! last-index xs x = x
by(simp add:last-index-def index-size-conv Let-def rev-nth[symmetric])

lemma index-eq-index-conv[simp]: x ∈ set xs ∨ y ∈ set xs ⟹
  (index xs x = index xs y) = (x = y)
by (induct xs) auto

lemma last-index-eq-index-conv[simp]: x ∈ set xs ∨ y ∈ set xs ⟹
  (last-index xs x = last-index xs y) = (x = y)
by (induct xs) (auto simp:last-index-Cons)

lemma inj-on-index: inj-on (index xs) (set xs)
by (simp add:inj-on-def)

lemma inj-on-last-index: inj-on (last-index xs) (set xs)
by (simp add:inj-on-def)

lemma index-conv-takeWhile: index xs x = size(takeWhile (λy. x≠y) xs)
by(induct xs) auto

lemma index-take: index xs x >= i ⟹ x ∉ set(take i xs)
apply(subst (asm) index-conv-takeWhile)
apply(subgoal-tac set(take i xs) <= set(takeWhile (op ≠ x) xs))
 apply(blast dest: set-takeWhileD)
apply(metis set-take-subset-set-take takeWhile-eq-take)
done

lemma last-index-drop:
  last-index xs x < i ⟹ x ∉ set(drop i xs)
apply(subgoal-tac set(drop i xs) = set(take (size xs − i) (rev xs)))
 apply(simp add: last-index-def index-take Let-def split:split-if-asm)
apply (metis rev-drop set-rev)
done

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*: $x \in set\ xs \Longrightarrow hidden\ (xs\ @\ [x])\ (last\text{-}index\ xs\ x)$
apply(*auto simp add*: *hidden-def nth-append rev-nth*[*symmetric*])
apply(*drule last-index-less*[*OF - le-refl*])
apply *simp*
done

lemma *hidden-inacc*: $hidden\ xs\ i \Longrightarrow last\text{-}index\ xs\ x \neq i$
by(*auto simp add*: *hidden-def last-index-drop last-index-less-size-conv*)

lemma [*simp*]: $hidden\ xs\ i \Longrightarrow hidden\ (xs@[x])\ i$
by(*auto simp add*:*hidden-def nth-append*)

lemma *fun-upds-apply*:
$(m(xs[\mapsto]ys))\ x =$
$(let\ xs' = take\ (size\ ys)\ xs$
$in\ if\ x \in set\ xs'\ then\ Some(ys\ !\ last\text{-}index\ xs'\ x)\ else\ m\ x)$
apply(*induct xs arbitrary*: *m ys*)
apply (*simp add*: *Let-def*)
apply(*case-tac ys*)
apply (*simp add*:*Let-def*)
apply (*simp add*: *Let-def last-index-Cons*)
done

lemma *map-upds-apply-eq-Some*:
$((m(xs[\mapsto]ys))\ x = Some\ y) =$
$(let\ xs' = take\ (size\ ys)\ xs$
$in\ if\ x \in set\ xs'\ then\ ys\ !\ last\text{-}index\ xs'\ x = y\ else\ m\ x = Some\ y)$
by(*simp add*:*fun-upds-apply Let-def*)

lemma *map-upds-upd-conv-last-index*:
$[\![x \in set\ xs;\ size\ xs \leq size\ ys\ ]\!]$
$\Longrightarrow m(xs[\mapsto]ys)(x\mapsto y) = m(xs[\mapsto]ys[last\text{-}index\ xs\ x := y])$
apply(*rule ext*)
apply(*simp add*:*fun-upds-apply eq-sym-conv Let-def*)
done

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* ⇒ $expr_1$ **where**
fin_1(*Val v*) = *Val v*
| fin_1(*throw e*) = *throw*(fin_1 *e*)

lemma *comp-final*: *final e* ⟹ $compE_1$ *Vs e* = fin_1 *e*

lemma [*simp*]:
⋀*Vs*. *max-vars* ($compE_1$ *Vs e*) = *max-vars e*
and ⋀*Vs*. *max-varss* ($compEs_1$ *Vs es*) = *max-varss es*

Compiling programs:

definition $compP_1$ :: *J-prog* ⇒ J_1*-prog*
where
$compP_1$ ≡ *compP* (λ(*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**
$unmod\ (new\ C)\ i = True\ |$
$unmod\ (Cast\ C\ e)\ i = unmod\ e\ i\ |$
$unmod\ (Val\ v)\ i = True\ |$
$unmod\ (e_1 \ll bop \gg e_2)\ i = (unmod\ e_1\ i \wedge unmod\ e_2\ i)\ |$
$unmod\ (Var\ i)\ j = True\ |$
$unmod\ (i{:=}e)\ j = (i \neq j \wedge unmod\ e\ j)\ |$
$unmod\ (e{\cdot}F\{D\})\ i = unmod\ e\ i\ |$
$unmod\ (e_1{\cdot}F\{D\}{:=}e_2)\ i = (unmod\ e_1\ i \wedge unmod\ e_2\ i)\ |$
$unmod\ (e{\cdot}M(es))\ i = (unmod\ e\ i \wedge unmods\ es\ i)\ |$
$unmod\ \{j{:}T;\ e\}\ i = unmod\ e\ i\ |$
$unmod\ (e_1;;e_2)\ i = (unmod\ e_1\ i \wedge\ \ unmod\ e_2\ i)\ |$
$unmod\ (if\ (e)\ e_1\ else\ e_2)\ i = (unmod\ e\ i \wedge unmod\ e_1\ i \wedge unmod\ e_2\ i)\ |$
$unmod\ (while\ (e)\ c)\ i = (unmod\ e\ i \wedge unmod\ c\ i)\ |$
$unmod\ (throw\ e)\ i = unmod\ e\ i\ |$
$unmod\ (try\ e_1\ catch(C\ i)\ e_2)\ j = (unmod\ e_1\ j \wedge (if\ i{=}j\ then\ False\ else\ unmod\ e_2\ j))\ |$

$unmods\ ([])\ i = True\ |$
$unmods\ (e\#es)\ i = (unmod\ e\ i \wedge unmods\ es\ i)$

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 $compE_1\text{-}pres\text{-}wt$: $\bigwedge Vs\ Ts\ U.$
$[\![\ P,[Vs[\mapsto]Ts] \vdash e :: U;\ size\ Ts = size\ Vs\ ]\!]$
$\Longrightarrow compP\ f\ P,Ts \vdash_1 compE_1\ \ Vs\ e :: U$
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 $image\text{-}last\text{-}index$: $A \subseteq set(xs@[x]) \Longrightarrow last\text{-}index\ (xs\ @\ [x])\ \text{`}\ A =$
$(if\ x \in A\ then\ insert\ (size\ xs)\ (last\text{-}index\ xs\ \text{`}\ (A-\{x\}))\ else\ last\text{-}index\ xs\ \text{`}\ A)$

lemma $A\text{-}compE_1\text{-}None[simp]$:
$\bigwedge Vs.\ \mathcal{A}\ e = None \Longrightarrow \mathcal{A}\ (compE_1\ \ Vs\ e) = None$
and $\bigwedge Vs.\ \mathcal{A}s\ es = None \Longrightarrow \mathcal{A}s\ (compEs_1\ \ Vs\ es) = None$

lemma $A\text{-}compE_1$:
$\bigwedge A\ \ Vs.\ [\![\ \mathcal{A}\ e = \lfloor A\rfloor;\ fv\ e \subseteq set\ Vs\ ]\!] \Longrightarrow \mathcal{A}\ (compE_1\ \ Vs\ e) = \lfloor last\text{-}index\ Vs\ \text{`}\ A\rfloor$
and $\bigwedge A\ \ Vs.\ [\![\ \mathcal{A}s\ es = \lfloor A\rfloor;\ fvs\ es \subseteq set\ Vs\ ]\!] \Longrightarrow \mathcal{A}s\ (compEs_1\ \ Vs\ es) = \lfloor last\text{-}index\ Vs\ \text{`}\ A\rfloor$

lemma $D\text{-}None[iff]$: $\mathcal{D}\ (e::'a\ exp)\ None$ **and** $[iff]$: $\mathcal{D}s\ (es::'a\ exp\ list)\ None$

lemma $D\text{-}last\text{-}index\text{-}compE_1$:
$\bigwedge A\ \ Vs.\ [\![\ A \subseteq set\ Vs;\ fv\ e \subseteq set\ Vs\ ]\!] \Longrightarrow$
$\mathcal{D}\ e\ \lfloor A\rfloor \Longrightarrow \mathcal{D}\ (compE_1\ \ Vs\ e)\ \lfloor last\text{-}index\ Vs\ \text{`}\ A\rfloor$
and $\bigwedge A\ \ Vs.\ [\![\ A \subseteq set\ Vs;\ fvs\ es \subseteq set\ Vs\ ]\!] \Longrightarrow$
$\mathcal{D}s\ es\ \lfloor A\rfloor \Longrightarrow \mathcal{D}s\ (compEs_1\ \ Vs\ es)\ \lfloor last\text{-}index\ Vs\ \text{`}\ A\rfloor$

lemma $last\text{-}index\text{-}image\text{-}set$: $distinct\ xs \Longrightarrow last\text{-}index\ xs\ \text{`}\ set\ xs = \{..<size\ xs\}$

lemma $D\text{-}compE_1$:
$[\![\ \mathcal{D}\ e\ \lfloor set\ Vs\rfloor;\ fv\ e \subseteq set\ Vs;\ distinct\ Vs\ ]\!] \Longrightarrow \mathcal{D}\ (compE_1\ \ Vs\ e)\ \lfloor\{..<length\ Vs\}\rfloor$

lemma $D\text{-}compE_1{}'$:
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 $compP_1\text{-}pres\text{-}wf$: $wf\text{-}J\text{-}prog\ P \Longrightarrow wf\text{-}J_1\text{-}prog\ (compP_1\ P)$

end

5.6 Compilation Stage 2

```
theory Compiler2
imports PCompiler J1 ../JVM/JVMExec
begin

primrec compE2 :: expr1 ⇒ instr list
  and compEs2 :: expr1 list ⇒ instr list where
  compE2 (new C) = [New C]
| compE2 (Cast C e) = compE2 e @ [Checkcast C]
| compE2 (Val v) = [Push v]
| compE2 (e1 «bop» e2) = compE2 e1 @ compE2 e2 @
  (case bop of Eq ⇒ [CmpEq]
           | Add ⇒ [IAdd])
| compE2 (Var i) = [Load i]
| compE2 (i:=e) = compE2 e @ [Store i, Push Unit]
| compE2 (e•F{D}) = compE2 e @ [Getfield F D]
| compE2 (e1•F{D} := e2) =
      compE2 e1 @ compE2 e2 @ [Putfield F D, Push Unit]
| compE2 (e•M(es)) = compE2 e @ compEs2 es @ [Invoke M (size es)]
| compE2 ({i:T; e}) = compE2 e
| compE2 (e1;;e2) = compE2 e1 @ [Pop] @ compE2 e2
| compE2 (if (e) e1 else e2) =
       (let cnd   = compE2 e;
            thn   = compE2 e1;
            els   = compE2 e2;
            test  = IfFalse (int(size thn + 2));
            thnex = Goto (int(size els + 1))
        in cnd @ [test] @ thn @ [thnex] @ els)
| compE2 (while (e) c) =
       (let cnd   = compE2 e;
            bdy   = compE2 c;
            test  = IfFalse (int(size bdy + 3));
            loop  = Goto (-int(size bdy + size cnd + 2))
        in cnd @ [test] @ bdy @ [Pop] @ [loop] @ [Push Unit])
| compE2 (throw e) = compE2 e @ [instr.Throw]
| compE2 (try e1 catch(C i) e2) =
   (let catch = compE2 e2
    in compE2 e1 @ [Goto (int(size catch)+2), Store i] @ catch)

| 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
```

| *compx*E_2 (*e*∙*F*{*D*}) *pc d* = *compx*E_2 *e pc d*
| *compx*E_2 (e_1∙*F*{*D*} := e_2) *pc d* =
compxE_2 e_1 *pc d* @ *compx*E_2 e_2 (*pc* + *size*(*comp*E_2 e_1)) (*d*+*1*)
| *compx*E_2 (*e*∙*M*(*es*)) *pc d* =
compxE_2 *e pc d* @ *compx*Es_2 *es* (*pc* + *size*(*comp*E_2 *e*)) (*d*+*1*)
| *compx*E_2 ({*i*:*T*; *e*}) *pc d* = *compx*E_2 *e pc d*
| *compx*E_2 (e_1;;e_2) *pc d* =
compxE_2 e_1 *pc d* @ *compx*E_2 e_2 (*pc*+*size*(*comp*E_2 e_1)+*1*) *d*
| *compx*E_2 (*if* (*e*) e_1 *else* e_2) *pc d* =
(*let* pc_1 = *pc* + *size*(*comp*E_2 *e*) + *1*;
pc_2 = pc_1 + *size*(*comp*E_2 e_1) + *1*
in *compx*E_2 *e pc d* @ *compx*E_2 e_1 pc_1 *d* @ *compx*E_2 e_2 pc_2 *d*)
| *compx*E_2 (*while* (*b*) *e*) *pc d* =
compxE_2 *b pc d* @ *compx*E_2 *e* (*pc*+*size*(*comp*E_2 *b*)+*1*) *d*
| *compx*E_2 (*throw e*) *pc d* = *compx*E_2 *e pc d*
| *compx*E_2 (*try* e_1 *catch*(*C i*) e_2) *pc d* =
(*let* pc_1 = *pc* + *size*(*comp*E_2 e_1)
in *compx*E_2 e_1 *pc d* @ *compx*E_2 e_2 (pc_1+*2*) *d* @ [(*pc*,pc_1,*C*,pc_1+*1*,*d*)])

| *compx*Es_2 [] *pc d* = []
| *compx*Es_2 (*e*#*es*) *pc d* = *compx*E_2 *e pc d* @ *compx*Es_2 *es* (*pc*+*size*(*comp*E_2 *e*)) (*d*+*1*)

primrec *max-stack* :: $expr_1$ ⇒ *nat*
and *max-stacks* :: $expr_1$ *list* ⇒ *nat* **where**
max-stack (*new C*) = *1*
| *max-stack* (*Cast C e*) = *max-stack e*
| *max-stack* (*Val v*) = *1*
| *max-stack* (e_1 ≪*bop*≫ e_2) = *max* (*max-stack* e_1) (*max-stack* e_2) + *1*
| *max-stack* (*Var i*) = *1*
| *max-stack* (*i*:=*e*) = *max-stack e*
| *max-stack* (*e*∙*F*{*D*}) = *max-stack e*
| *max-stack* (e_1∙*F*{*D*} := e_2) = *max* (*max-stack* e_1) (*max-stack* e_2) + *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 *comp*Mb_2 :: $expr_1$ ⇒ *jvm-method*
where
compMb_2 ≡ λ*body*.
let ins = *comp*E_2 *body* @ [*Return*];
xt = *compx*E_2 *body 0 0*
in (*max-stack body*, *max-vars body*, *ins*, *xt*)

definition *comp*P_2 :: J_1-*prog* ⇒ *jvm-prog*

where
$compP_2 \equiv compP\ compMb_2$

lemma $compMb_2$ [*simp*]:
$compMb_2\ e = (max\text{-}stack\ e,\ max\text{-}vars\ e,\ compE_2\ e\ @\ [Return],\ compxE_2\ e\ 0\ 0)$

end

5.7 Correctness of Stage 2

theory *Correctness2*
imports ~~/*src*/*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 $before :: jvm\text{-}prog \Rightarrow cname \Rightarrow mname \Rightarrow nat \Rightarrow instr\ list \Rightarrow bool$
$((\text{-},\text{-},\text{-},\text{-}/ \rhd \text{-})\ [51,0,0,0,51]\ 50)$ **where**
$P,C,M,pc \rhd is \longleftrightarrow prefixeq\ is\ (drop\ pc\ (instrs\text{-}of\ P\ C\ M))$

definition $at :: jvm\text{-}prog \Rightarrow cname \Rightarrow mname \Rightarrow nat \Rightarrow instr \Rightarrow bool$
$((\text{-},\text{-},\text{-},\text{-}/ \triangleright \text{-})\ [51,0,0,0,51]\ 50)$ **where**
$P,C,M,pc \triangleright i \longleftrightarrow (\exists is.\ drop\ pc\ (instrs\text{-}of\ P\ C\ M) = i\#is)$

lemma [*simp*]: $P,C,M,pc \rhd []$

lemma [*simp*]: $P,C,M,pc \rhd (i\#is) = (P,C,M,pc \triangleright i \wedge P,C,M,pc + 1 \rhd is)$

lemma [*simp*]: $P,C,M,pc \rhd (is_1\ @\ is_2) = (P,C,M,pc \rhd is_1 \wedge P,C,M,pc + size\ is_1 \rhd is_2)$

lemma [*simp*]: $P,C,M,pc \triangleright i \Longrightarrow instrs\text{-}of\ P\ C\ M\ !\ pc = i$

lemma *beforeM*:
$P \vdash C\ sees\ M\colon Ts \rightarrow T = body\ in\ D \Longrightarrow$
$compP_2\ P,D,M,0 \rhd compE_2\ body\ @\ [Return]$

This lemma executes a single instruction by rewriting:

lemma [*simp*]:
$P,C,M,pc \triangleright instr \Longrightarrow$
$(P \vdash (None,\ h,\ (vs,ls,C,M,pc)\ \#\ frs)\ -jvm\rightarrow \sigma') =$
$((None,\ h,\ (vs,ls,C,M,pc)\ \#\ frs) = \sigma' \vee$
$(\exists \sigma.\ exec(P,(None,\ h,\ (vs,ls,C,M,pc)\ \#\ frs)) = Some\ \sigma \wedge P \vdash \sigma\ -jvm\rightarrow \sigma'))$

5.7.2 Exception tables

definition $pcs :: ex\text{-}table \Rightarrow nat\ set$
where
$pcs\ xt\ \equiv\ \bigcup (f,t,C,h,d) \in set\ xt.\ \{f\ ..<\ t\}$

lemma *pcs-subset*:
shows $\bigwedge pc\ d.\ pcs(compxE_2\ e\ pc\ d) \subseteq \{pc..<pc+size(compE_2\ e)\}$
and $\bigwedge pc\ d.\ pcs(compxEs_2\ es\ pc\ d) \subseteq \{pc..<pc+size(compEs_2\ es)\}$

lemma [*simp*]: $pcs\ [] = \{\}$

lemma [*simp*]: $pcs\ (x\#xt) = \{fst\ x\ ..<\ fst(snd\ x)\} \cup pcs\ xt$

lemma [*simp*]: $pcs(xt_1\ @\ xt_2) = pcs\ xt_1 \cup pcs\ xt_2$

lemma [*simp*]: $pc < pc_0 \lor pc_0{+}\mathit{size}(\mathit{compE}_2\ e) \le pc \Longrightarrow pc \notin \mathit{pcs}(\mathit{compxE}_2\ e\ pc_0\ d)$

lemma [*simp*]: $pc < pc_0 \lor pc_0{+}\mathit{size}(\mathit{compEs}_2\ es) \le pc \Longrightarrow pc \notin \mathit{pcs}(\mathit{compxEs}_2\ es\ pc_0\ d)$

lemma [*simp*]: $pc_1 + \mathit{size}(\mathit{compE}_2\ e_1) \le pc_2 \Longrightarrow \mathit{pcs}(\mathit{compxE}_2\ e_1\ pc_1\ d_1) \cap \mathit{pcs}(\mathit{compxE}_2\ e_2\ pc_2\ d_2) = \{\}$

lemma [*simp*]: $pc_1 + \mathit{size}(\mathit{compE}_2\ e) \le pc_2 \Longrightarrow \mathit{pcs}(\mathit{compxE}_2\ e\ pc_1\ d_1) \cap \mathit{pcs}(\mathit{compxEs}_2\ es\ pc_2\ d_2) = \{\}$

lemma [*simp*]:
$pc \notin \mathit{pcs}\ xt_0 \Longrightarrow \mathit{match\text{-}ex\text{-}table}\ P\ C\ pc\ (xt_0\ @\ xt_1) = \mathit{match\text{-}ex\text{-}table}\ P\ C\ pc\ xt_1$

lemma [*simp*]: $[\![\ x \in \mathit{set}\ xt;\ pc \notin \mathit{pcs}\ xt\ ]\!] \Longrightarrow \neg\ \mathit{matches\text{-}ex\text{-}entry}\ P\ D\ pc\ x$

lemma [*simp*]:
assumes *xe*: $xe \in \mathit{set}(\mathit{compxE}_2\ e\ pc\ d)$ **and** *outside*: $pc' < pc \lor pc{+}\mathit{size}(\mathit{compE}_2\ e) \le pc'$
shows $\neg\ \mathit{matches\text{-}ex\text{-}entry}\ P\ C\ pc'\ xe$

lemma [*simp*]:
assumes *xe*: $xe \in \mathit{set}(\mathit{compxEs}_2\ es\ pc\ d)$ **and** *outside*: $pc' < pc \lor pc{+}\mathit{size}(\mathit{compEs}_2\ es) \le pc'$
shows $\neg\ \mathit{matches\text{-}ex\text{-}entry}\ P\ C\ pc'\ xe$

lemma *match-ex-table-app*[*simp*]:
$\forall xte \in \mathit{set}\ xt_1.\ \neg\ \mathit{matches\text{-}ex\text{-}entry}\ P\ D\ pc\ xte \Longrightarrow$
$\mathit{match\text{-}ex\text{-}table}\ P\ D\ pc\ (xt_1\ @\ xt) = \mathit{match\text{-}ex\text{-}table}\ P\ D\ pc\ xt$

lemma [*simp*]:
$\forall x \in \mathit{set}\ xtab.\ \neg\ \mathit{matches\text{-}ex\text{-}entry}\ P\ C\ pc\ x \Longrightarrow$
$\mathit{match\text{-}ex\text{-}table}\ P\ C\ pc\ xtab = \mathit{None}$

lemma *match-ex-entry*:
$\mathit{matches\text{-}ex\text{-}entry}\ P\ C\ pc\ (\mathit{start},\ \mathit{end},\ \mathit{catch\text{-}type},\ \mathit{handler}) =$
$(\mathit{start} \le pc \land pc < \mathit{end} \land\ P \vdash C \preceq^* \mathit{catch\text{-}type})$

definition *caught* :: $\mathit{jvm\text{-}prog} \Rightarrow pc \Rightarrow \mathit{heap} \Rightarrow \mathit{addr} \Rightarrow \mathit{ex\text{-}table} \Rightarrow \mathit{bool}$ **where**
$\mathit{caught}\ P\ pc\ h\ a\ xt \longleftrightarrow$
$(\exists\, \mathit{entry} \in \mathit{set}\ xt.\ \mathit{matches\text{-}ex\text{-}entry}\ P\ (\mathit{cname\text{-}of}\ h\ a)\ pc\ \mathit{entry})$

definition *beforex* :: $\mathit{jvm\text{-}prog} \Rightarrow \mathit{cname} \Rightarrow \mathit{mname} \Rightarrow \mathit{ex\text{-}table} \Rightarrow \mathit{nat\ set} \Rightarrow \mathit{nat} \Rightarrow \mathit{bool}$
((2-,/-,/- ▷/ - /′/ -,/-) [51,0,0,0,0,51] 50) **where**
$P,C,M \rhd xt\ /\ I,d \longleftrightarrow$
$(\exists\, xt_0\ xt_1.\ \mathit{ex\text{-}table\text{-}of}\ P\ C\ M = xt_0\ @\ xt\ @\ xt_1 \land \mathit{pcs}\ xt_0 \cap I = \{\} \land \mathit{pcs}\ xt \subseteq I \land$
$(\forall\, pc \in I.\ \forall\, C\ pc'\ d'.\ \mathit{match\text{-}ex\text{-}table}\ P\ C\ pc\ xt_1 = \lfloor(pc',d')\rfloor \longrightarrow d' \le d))$

definition *dummyx* :: $\mathit{jvm\text{-}prog} \Rightarrow \mathit{cname} \Rightarrow \mathit{mname} \Rightarrow \mathit{ex\text{-}table} \Rightarrow \mathit{nat\ set} \Rightarrow \mathit{nat} \Rightarrow \mathit{bool}$ *((2-,-,- ▹/ - ′/-,-) [51,0,0,0,0,51] 50)* **where**
$P,C,M \triangleright xt/I,d \longleftrightarrow P,C,M \rhd xt/I,d$

lemma *beforexD1*: $P,C,M \rhd xt\ /\ I,d \Longrightarrow \mathit{pcs}\ xt \subseteq I$

lemma *beforex-mono*: $[\![\ P,C,M \rhd xt/I,d';\ d' \le d\ ]\!] \Longrightarrow P,C,M \rhd xt/I,d$

lemma [*simp*]: $P,C,M \rhd xt/I,d \Longrightarrow P,C,M \rhd xt/I,\mathit{Suc}\ d$

lemma *beforex-append*[*simp*]:
$pcs\ xt_1 \cap pcs\ xt_2 = \{\} \Longrightarrow$
$P,C,M \rhd xt_1 \ @\ xt_2/I,d =$
$(P,C,M \rhd xt_1/I{-}pcs\ xt_2,d\ \wedge\ P,C,M \rhd xt_2/I{-}pcs\ xt_1,d \wedge P,C,M \rhd xt_1@xt_2/I,d)$

lemma *beforex-appendD1*:
$[\![\ P,C,M \rhd xt_1 \ @\ xt_2 \ @\ [(f,t,D,h,d)]\ /\ I,d;$
$pcs\ xt_1 \subseteq J;\ J \subseteq I;\ J \cap pcs\ xt_2 = \{\}\ ]\!]$
$\Longrightarrow P,C,M \rhd xt_1\ /\ J,d$

lemma *beforex-appendD2*:
$[\![\ P,C,M \rhd xt_1 \ @\ xt_2 \ @\ [(f,t,D,h,d)]\ /\ I,d;$
$pcs\ xt_2 \subseteq J;\ J \subseteq I;\ J \cap pcs\ xt_1 = \{\}\ ]\!]$
$\Longrightarrow P,C,M \rhd xt_2\ /\ J,d$

lemma *beforexM*:
$P \vdash C\ sees\ M\colon Ts{\rightarrow}T = body\ in\ D \Longrightarrow$
$compP_2\ P,D,M \rhd compxE_2\ body\ 0\ 0/\{..{<}size(compE_2\ body)\},0$

lemma *match-ex-table-SomeD2*:
$[\![\ match\text{-}ex\text{-}table\ P\ D\ pc\ (ex\text{-}table\text{-}of\ P\ C\ M) = \lfloor(pc',d')\rfloor;$
$P,C,M \rhd xt/I,d;\ \forall x \in set\ xt.\ \neg\ matches\text{-}ex\text{-}entry\ P\ D\ pc\ x;\ pc \in I\ ]\!]$
$\Longrightarrow d' \le d$

lemma *match-ex-table-SomeD1*:
$[\![\ match\text{-}ex\text{-}table\ P\ D\ pc\ (ex\text{-}table\text{-}of\ P\ C\ M) = \lfloor(pc',d')\rfloor;$
$P,C,M \rhd xt\ /\ I,d;\ pc \in I;\ pc \notin pcs\ xt\ ]\!] \Longrightarrow d' \le d$

5.7.3 The correctness proof

definition
$handle :: jvm\text{-}prog \Rightarrow cname \Rightarrow mname \Rightarrow addr \Rightarrow heap \Rightarrow val\ list \Rightarrow val\ list \Rightarrow nat \Rightarrow frame\ list$
$\Rightarrow jvm\text{-}state$ **where**
$handle\ P\ C\ M\ a\ h\ vs\ ls\ pc\ frs = find\text{-}handler\ P\ a\ h\ ((vs,ls,C,M,pc)\ \#\ frs)$

lemma *handle-Cons*:
$[\![\ P,C,M \rhd xt/I,d;\ d \le size\ vs;\ pc \in I;$
$\forall x \in set\ xt.\ \neg\ matches\text{-}ex\text{-}entry\ P\ (cname\text{-}of\ h\ xa)\ pc\ x\ ]\!] \Longrightarrow$
$handle\ P\ C\ M\ xa\ h\ (v\ \#\ vs)\ ls\ pc\ frs = handle\ P\ C\ M\ xa\ h\ vs\ ls\ pc\ frs$
lemma *handle-append*:
$[\![\ P,C,M \rhd xt/I,d;\ d \le size\ vs;\ pc \in I;\ pc \notin pcs\ xt\ ]\!] \Longrightarrow$
$handle\ P\ C\ M\ xa\ h\ (ws\ @\ vs)\ ls\ pc\ frs = handle\ P\ C\ M\ xa\ h\ vs\ ls\ pc\ frs$

lemma *aux-isin*[*simp*]: $[\![\ B \subseteq A;\ a \in B\ ]\!] \Longrightarrow a \in A$

lemma fixes P_1 **defines** [*simp*]: $P \equiv compP_2\ P_1$
shows *Jcc*:
$P_1 \vdash_1 \langle e,(h_0,ls_0)\rangle \Rightarrow \langle ef,(h_1,ls_1)\rangle \Longrightarrow$
$(\bigwedge C\ M\ pc\ v\ xa\ vs\ frs\ I.$
$[\![\ P,C,M,pc \rhd compE_2\ e;\ P,C,M \rhd compxE_2\ e\ pc\ (size\ vs)/I,size\ vs;$
$\{pc..{<}pc{+}size(compE_2\ e)\} \subseteq I\ ]\!] \Longrightarrow$

$(ef = Val\ v \longrightarrow$
$P \vdash (None,h_0,(vs,ls_0,C,M,pc)\#frs)\ -jvm\rightarrow$
$(None,h_1,(v\#vs,ls_1,C,M,pc{+}size(compE_2\ e))\#frs))$
$\wedge$
$(ef = Throw\ xa \longrightarrow$
$(\exists\, pc_1.\ pc \leq pc_1 \wedge pc_1 < pc + size(compE_2\ e) \wedge$
$\neg\ caught\ P\ pc_1\ h_1\ xa\ (compxE_2\ e\ pc\ (size\ vs)) \wedge$
$P \vdash (None,h_0,(vs,ls_0,C,M,pc)\#frs)\ -jvm\rightarrow\ handle\ P\ C\ M\ xa\ h_1\ vs\ ls_1\ pc_1\ frs)))$
and $P_1 \vdash_1 \langle es,(h_0,ls_0)\rangle\ [\Rightarrow]\ \langle fs,(h_1,ls_1)\rangle \Longrightarrow$
$(\bigwedge C\ M\ pc\ ws\ xa\ es'\ vs\ frs\ I.$
$[\![\ P,C,M,pc \rhd compEs_2\ es;\ P,C,M \rhd compxEs_2\ es\ pc\ (size\ vs)/I,size\ vs;$
$\{pc..{<}pc{+}size(compEs_2\ es)\} \subseteq I\ ]\!] \Longrightarrow$
$(fs = map\ Val\ ws \longrightarrow$
$P \vdash (None,h_0,(vs,ls_0,C,M,pc)\#frs)\ -jvm\rightarrow$
$(None,h_1,(rev\ ws\ @\ vs,ls_1,C,M,pc{+}size(compEs_2\ es))\#frs))$
$\wedge$
$(fs = map\ Val\ ws\ @\ Throw\ xa\ \#\ es' \longrightarrow$
$(\exists\, pc_1.\ pc \leq pc_1 \wedge pc_1 < pc + size(compEs_2\ es) \wedge$
$\neg\ caught\ P\ pc_1\ h_1\ xa\ (compxEs_2\ es\ pc\ (size\ vs)) \wedge$
$P \vdash (None,h_0,(vs,ls_0,C,M,pc)\#frs)\ -jvm\rightarrow\ handle\ P\ C\ M\ xa\ h_1\ vs\ ls_1\ pc_1\ frs)))$

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* ⊢ *C sees M*:*Ts*→*T* = (*pns*,*body*) *in C*
and *eval*: *P* ⊢ ⟨*body*,(*h*,[*this*#*pns* [↦] *vs*])⟩ ⇒ ⟨*e*′,(*h*′,*l*′)⟩
and *sizes*: *size vs* = *size pns* + *1* *size rest* = *max-vars body*
shows *J2JVM P* ⊢ (*None*,*h*,[([],*vs*@*rest*,*C*,*M*,*0*)]) −*jvm*→ (*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*:
 $set\ E \subseteq types\ P \Longrightarrow ty_l\ E\ A \in list\ mxl\ (err\ (types\ P))$
locale *TC1* = *TC0*
begin

primrec $compT :: ty\ list \Rightarrow nat\ hyperset \Rightarrow ty\ list \Rightarrow expr_1 \Rightarrow ty_i'\ list$ **and**
 $compTs :: ty\ list \Rightarrow nat\ hyperset \Rightarrow ty\ list \Rightarrow expr_1\ list \Rightarrow ty_i'\ list$ **where**
$compT\ E\ A\ ST\ (new\ C) = []$

| *compT E A ST (Cast C e) =*
compT E A ST e @ [after E A ST e]
| *compT E A ST (Val v) = []*
| *compT E A ST* $(e_1 \ll bop \gg e_2)$ =
$(let\ ST_1 = ty\ E\ e_1\#ST;\ A_1 = A \sqcup \mathcal{A}\ e_1\ in$
$compT\ E\ A\ ST\ e_1\ @\ [after\ E\ A\ ST\ e_1]\ @$
$compT\ E\ A_1\ ST_1\ e_2\ @\ [after\ E\ A_1\ ST_1\ e_2])$
| *compT E A ST (Var i) = []*
| *compT E A ST (i := e) = compT E A ST e @*
$[after\ E\ A\ ST\ e,\ ty_i{}'\ ST\ E\ (A \sqcup \mathcal{A}\ e \sqcup \lfloor\{i\}\rfloor)]$
| *compT E A ST (e•F{D}) =*
compT E A ST e @ [after E A ST e]
| $compT\ E\ A\ ST\ (e_1{\cdot}F\{D\} := e_2) =$
$(let\ ST_1 = ty\ \ E\ e_1\#ST;\ A_1 = A \sqcup \mathcal{A}\ e_1;\ A_2 = A_1 \sqcup \mathcal{A}\ e_2\ in$
$compT\ E\ A\ ST\ e_1\ @\ [after\ E\ A\ ST\ e_1]\ @$
$compT\ E\ A_1\ ST_1\ e_2\ @\ [after\ E\ A_1\ ST_1\ e_2]\ @$
$[ty_i{}'\ ST\ E\ A_2])$
| $compT\ E\ A\ ST\ \{i{:}T;\ e\} = compT\ (E@[T])\ (A \ominus i)\ ST\ e$
| $compT\ E\ A\ ST\ (e_1;;e_2) =$
$(let\ A_1 = A \sqcup \mathcal{A}\ e_1\ in$
$compT\ E\ A\ ST\ e_1\ @\ [after\ E\ A\ ST\ e_1,\ ty_i{}'\ ST\ E\ A_1]\ @$
$compT\ E\ A_1\ ST\ e_2)$
| $compT\ E\ A\ ST\ (if\ (e)\ e_1\ else\ e_2) =$
$(let\ A_0 = A \sqcup \mathcal{A}\ e;\ \tau = ty_i{}'\ ST\ E\ A_0\ in$
$compT\ E\ A\ ST\ e\ @\ [after\ E\ A\ ST\ e,\ \tau]\ @$
$compT\ E\ A_0\ ST\ e_1\ @\ [after\ E\ A_0\ ST\ e_1,\ \tau]\ @$
$compT\ E\ A_0\ ST\ e_2)$
| *compT E A ST (while (e) c) =*
$(let\ A_0 = A \sqcup \mathcal{A}\ e;\ \ A_1 = A_0 \sqcup \mathcal{A}\ c;\ \tau = ty_i{}'\ ST\ E\ A_0\ in$
$compT\ E\ A\ ST\ e\ @\ [after\ E\ A\ ST\ e,\ \tau]\ @$
$compT\ E\ A_0\ ST\ c\ @\ [after\ E\ A_0\ ST\ c,\ ty_i{}'\ ST\ E\ A_1,\ ty_i{}'\ ST\ E\ A_0])$
| *compT E A ST (throw e) = compT E A ST e @ [after E A ST e]*
| *compT E A ST (e•M(es)) =*
compT E A ST e @ [after E A ST e] @
$compTs\ E\ (A \sqcup \mathcal{A}\ e)\ (ty\ \ E\ e\ \#\ ST)\ es$
| $compT\ E\ A\ ST\ (try\ e_1\ catch(C\ i)\ e_2) =$
$compT\ E\ A\ ST\ e_1\ @\ [after\ E\ A\ ST\ e_1]\ @$
$[ty_i{}'\ (Class\ C\#ST)\ E\ A,\ ty_i{}'\ ST\ (E@[Class\ C])\ (A \sqcup \lfloor\{i\}\rfloor)]\ @$
$compT\ (E@[Class\ C])\ (A \sqcup \lfloor\{i\}\rfloor)\ ST\ e_2$
| *compTs E A ST [] = []*
| *compTs E A ST (e#es) = compT E A ST e @ [after E A ST e] @*
$compTs\ E\ (A \sqcup (\mathcal{A}\ e))\ (ty\ E\ e\ \#\ ST)\ es$

definition $compT_a :: ty\ list \Rightarrow nat\ hyperset \Rightarrow ty\ list \Rightarrow expr_1 \Rightarrow ty_i{}'\ list$ **where**
$compT_a\ E\ A\ ST\ e = compT\ E\ A\ ST\ e\ @\ [after\ E\ A\ ST\ e]$

end

lemma $compE_2$*-not-Nil*[*simp*]: $compE_2\ e \neq []$
lemma (**in** *TC1*) *compT-sizes*[*simp*]:
shows $\bigwedge E\ A\ ST.\ size(compT\ E\ A\ ST\ e) = size(compE_2\ e) - 1$
and $\bigwedge E\ A\ ST.\ size(compTs\ E\ A\ ST\ es) = size(compEs_2\ es)$

lemma (**in** *TC1*) [*simp*]: $\bigwedge ST\ E.\ \lfloor\tau\rfloor \notin set\ (compT\ E\ None\ ST\ e)$

and [*simp*]: $\bigwedge ST\ E.\ \lfloor\tau\rfloor \notin set\ (compTs\ E\ None\ ST\ es)$

lemma (**in** *TC0*) *pair-eq-ty$_i$'-conv*:
$(\lfloor(ST,\ LT)\rfloor = ty_i'\ ST_0\ E\ A) =$
$(case\ A\ of\ None \Rightarrow False \mid Some\ A \Rightarrow (ST = ST_0 \wedge LT = ty_l\ E\ A))$

lemma (**in** *TC0*) *pair-conv-ty$_i$'*:
$\lfloor(ST,\ ty_l\ E\ A)\rfloor = ty_i'\ ST\ E\ \lfloor A\rfloor$

lemma (**in** *TC1*) *compT-LT-prefix*:
$\bigwedge E\ A\ ST_0.\ [\![\ \lfloor(ST,LT)\rfloor \in set(compT\ E\ A\ ST_0\ e);\ \mathcal{B}\ e\ (size\ E)\ ]\!]$
$\Longrightarrow P \vdash \lfloor(ST,LT)\rfloor \le'\ ty_i'\ ST\ E\ A$
and
$\bigwedge E\ A\ ST_0.\ [\![\ \lfloor(ST,LT)\rfloor \in set(compTs\ E\ A\ ST_0\ es);\ \mathcal{B}s\ es\ (size\ E)\ ]\!]$
$\Longrightarrow P \vdash \lfloor(ST,LT)\rfloor \le'\ ty_i'\ ST\ E\ A$

lemma [*iff*]: $OK\ None \in states\ P\ mxs\ mxl$
lemma (**in** *TC0*) *after-in-states*:
$[\![\ wf\text{-}prog\ p\ P;\ P,E \vdash_1 e :: T;\ set\ E \subseteq types\ P;\ set\ ST \subseteq types\ P;$
$size\ ST + max\text{-}stack\ e \le mxs\ ]\!]$
$\Longrightarrow OK\ (after\ E\ A\ ST\ e) \in states\ P\ mxs\ mxl$

lemma (**in** *TC0*) *OK-ty$_i$'-in-statesI*[*simp*]:
$[\![\ set\ E \subseteq types\ P;\ set\ ST \subseteq types\ P;\ size\ ST \le mxs\ ]\!]$
$\Longrightarrow OK\ (ty_i'\ ST\ E\ A) \in states\ P\ mxs\ mxl$

lemma *is-class-type-aux*: $is\text{-}class\ P\ C \Longrightarrow is\text{-}type\ P\ (Class\ C)$
theorem (**in** *TC1*) *compT-states*:
assumes *wf*: *wf-prog p P*
shows $\bigwedge E\ T\ A\ ST.$
$[\![\ P,E \vdash_1 e :: T;\ set\ E \subseteq types\ P;\ set\ ST \subseteq types\ P;$
$size\ ST + max\text{-}stack\ e \le mxs;\ size\ E + max\text{-}vars\ e \le mxl\ ]\!]$
$\Longrightarrow OK\ `\ set(compT\ E\ A\ ST\ e) \subseteq states\ P\ mxs\ mxl$
and $\bigwedge E\ Ts\ A\ ST.$
$[\![\ P,E \vdash_1 es[::]Ts;\ set\ E \subseteq types\ P;\ set\ ST \subseteq types\ P;$
$size\ ST + max\text{-}stacks\ es \le mxs;\ size\ E + max\text{-}varss\ es \le mxl\ ]\!]$
$\Longrightarrow OK\ `\ set(compTs\ E\ A\ ST\ es) \subseteq states\ P\ mxs\ mxl$

definition $shift :: nat \Rightarrow ex\text{-}table \Rightarrow ex\text{-}table$
where
$shift\ n\ xt \equiv map\ (\lambda(from,to,C,handler,depth).\ (from+n,to+n,C,handler+n,depth))\ xt$

lemma [*simp*]: $shift\ 0\ xt = xt$
lemma [*simp*]: $shift\ n\ [] = []$
lemma [*simp*]: $shift\ n\ (xt_1\ @\ xt_2) = shift\ n\ xt_1\ @\ shift\ n\ xt_2$
lemma [*simp*]: $shift\ m\ (shift\ n\ xt) = shift\ (m+n)\ xt$
lemma [*simp*]: $pcs\ (shift\ n\ xt) = \{pc+n|pc.\ pc \in pcs\ xt\}$

lemma *shift-compxE$_2$*:
shows $\bigwedge pc\ pc'\ d.\ shift\ pc\ (compxE_2\ e\ pc'\ d) = compxE_2\ e\ (pc' + pc)\ d$
and $\bigwedge pc\ pc'\ d.\ shift\ pc\ (compxEs_2\ es\ pc'\ d) = compxEs_2\ es\ (pc' + pc)\ d$

lemma *compxE$_2$-size-convs*[*simp*]:

shows $n \neq 0 \Longrightarrow compxE_2\ e\ n\ d = shift\ n\ (compxE_2\ e\ 0\ d)$
and $n \neq 0 \Longrightarrow compxEs_2\ es\ n\ d = shift\ n\ (compxEs_2\ es\ 0\ d)$
locale *TC2* = *TC1* +
  fixes T_r :: *ty* **and** *mxs* :: *pc*
begin

definition
  wt-instrs :: *instr list* ⇒ *ex-table* ⇒ $ty_i{}'$ *list* ⇒ *bool*
    ((⊢ -, - /[::]/ -) [0,0,51] 50) **where**
  ⊢ *is*,*xt* [::] $\tau s \longleftrightarrow size\ is < size\ \tau s \wedge pcs\ xt \subseteq \{0..<size\ is\} \wedge$
  $(\forall pc< size\ is.\ P,T_r,mxs,size\ \tau s,xt \vdash is!pc,pc :: \tau s)$

end

notation *TC2.wt-instrs* ((-,-,- ⊢/ -, - /[::]/ -) [50,50,50,50,50,51] 50)

lemma (**in** *TC2*) [*simp*]: $\tau s \neq [] \Longrightarrow \vdash [],[]\ [::]\ \tau s$
lemma [*simp*]: *eff i P pc et None* = []
lemma *wt-instr-appR*:
  $[\![\ P,T,m,mpc,xt \vdash is!pc,pc :: \tau s;$
    $pc < size\ is;\ size\ is < size\ \tau s;\ mpc \leq size\ \tau s;\ mpc \leq mpc'\ ]\!]$
  $\Longrightarrow P,T,m,mpc',xt \vdash is!pc,pc :: \tau s @ \tau s'$

lemma *relevant-entries-shift* [*simp*]:
  relevant-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) $\Longrightarrow$
    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*:
  $[\![\ P,T,m,mpc,xt \vdash i,pc :: \tau s;\ pc < size\ \tau s;\ mpc \leq size\ \tau s\ ]\!]$
  $\Longrightarrow P,T,m,mpc + size\ \tau s',shift\ (size\ \tau s')\ xt \vdash i,pc+size\ \tau s' :: \tau s' @ \tau s$

lemma *wt-instr-Cons*:
  $[\![\ P,T,m,mpc - 1,[] \vdash i,pc - 1 :: \tau s;$
    $0 < pc;\ 0 < mpc;\ pc < size\ \tau s + 1;\ mpc \leq size\ \tau s + 1\ ]\!]$
  $\Longrightarrow P,T,m,mpc,[] \vdash i,pc :: \tau \# \tau s$

lemma *wt-instr-append*:
  $[\![\ P,T,m,mpc - size\ \tau s',[] \vdash i,pc - size\ \tau s' :: \tau s;$
    $size\ \tau s' \leq pc;\ size\ \tau s' \leq mpc;\ pc < size\ \tau s + size\ \tau s';\ mpc \leq size\ \tau s + size\ \tau s'\ ]\!]$
  $\Longrightarrow P,T,m,mpc,[] \vdash i,pc :: \tau s' @ \tau s$

lemma *xcpt-app-pcs*:
  $pc \notin pcs\ xt \Longrightarrow xcpt\text{-}app\ i\ P\ pc\ mxs\ xt\ \tau$

lemma *xcpt-eff-pcs*:
$pc \notin pcs\ xt \Longrightarrow xcpt\text{-}eff\ i\ P\ pc\ \tau\ xt = []$

lemma *pcs-shift*:
$pc < n \Longrightarrow pc \notin pcs\ (shift\ n\ xt)$

lemma *wt-instr-appRx*:
$[\![\ P,T,m,mpc,xt \vdash is!pc,pc :: \tau s;\ pc < size\ is;\ size\ is < size\ \tau s;\ mpc \leq size\ \tau s\ ]\!]$
$\Longrightarrow P,T,m,mpc,xt\ @\ shift\ (size\ is)\ xt' \vdash is!pc,pc :: \tau s$

lemma *wt-instr-appLx*:
$[\![\ P,T,m,mpc,xt \vdash i,pc :: \tau s;\ pc \notin pcs\ xt'\ ]\!]$
$\Longrightarrow P,T,m,mpc,xt'@xt \vdash i,pc :: \tau s$

lemma (**in** *TC2*) *wt-instrs-extR*:
$\vdash is,xt\ [::]\ \tau s \Longrightarrow \vdash is,xt\ [::]\ \tau s\ @\ \tau s'$

lemma (**in** *TC2*) *wt-instrs-ext*:
$[\![\vdash is_1,xt_1\ [::]\ \tau s_1@\tau s_2;\ \vdash is_2,xt_2\ [::]\ \tau s_2;\ size\ \tau s_1 = size\ is_1\ ]\!]$
$\Longrightarrow \vdash is_1@is_2,\ xt_1\ @\ shift\ (size\ is_1)\ xt_2\ [::]\ \tau s_1@\tau s_2$
corollary (**in** *TC2*) *wt-instrs-ext2*:
$[\![\vdash is_2,xt_2\ [::]\ \tau s_2;\ \vdash is_1,xt_1\ [::]\ \tau s_1@\tau s_2;\ size\ \tau s_1 = size\ is_1\ ]\!]$
$\Longrightarrow \vdash is_1@is_2,\ xt_1\ @\ shift\ (size\ is_1)\ xt_2\ [::]\ \tau s_1@\tau s_2$

corollary (**in** *TC2*) *wt-instrs-ext-prefix* [*trans*]:
$[\![\vdash is_1,xt_1\ [::]\ \tau s_1@\tau s_2;\ \vdash is_2,xt_2\ [::]\ \tau s_3;$
$size\ \tau s_1 = size\ is_1;\ prefixeq\ \tau s_3\ \tau s_2\ ]\!]$
$\Longrightarrow \vdash is_1@is_2,\ xt_1\ @\ shift\ (size\ is_1)\ xt_2\ [::]\ \tau s_1@\tau s_2$

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*]:
$[\![\vdash is_2,xt_2\ [::]\ \tau\#\tau s_2;\ \vdash is_1,xt_1\ [::]\ \tau s_1;$
$last\ \tau s_1 = \tau;\ \ size\ \tau s_1 = size\ is_1{+}1\ ]\!]$
$\Longrightarrow \vdash is_1@is_2,\ xt_1@shift\ (size\ is_1)\ xt_2\ [::]\ \tau s_1@\tau s_2$

corollary (**in** *TC2*) *wt-instrs-append-last*[*trans*]:
$[\![\vdash is,xt\ [::]\ \tau s;\ P,T_r,mxs,mpc,[] \vdash i,pc :: \tau s;$
$pc = size\ is;\ mpc = size\ \tau s;\ size\ is + 1 < size\ \tau s\ ]\!]$
$\Longrightarrow \vdash is@[i],xt\ [::]\ \tau s$

corollary (**in** *TC2*) *wt-instrs-app2*:
$[\![\vdash is_2,xt_2\ [::]\ \tau'\#\tau s_2;\ \ \vdash is_1,xt_1\ [::]\ \tau\#\tau s_1@[\tau'];$
$xt' = xt_1\ @\ shift\ (size\ is_1)\ xt_2;\ \ size\ \tau s_1{+}1 = size\ is_1\ ]\!]$
$\Longrightarrow \vdash is_1@is_2,xt'\ [::]\ \tau\#\tau s_1@\tau'\#\tau s_2$

corollary (**in** *TC2*) *wt-instrs-app2-simp*[*trans,simp*]:
$[\![\vdash is_2,xt_2\ [::]\ \tau'\#\tau s_2;\ \ \vdash is_1,xt_1\ [::]\ \tau\#\tau s_1@[\tau'];\ size\ \tau s_1{+}1 = size\ is_1\ ]\!]$
$\Longrightarrow \vdash is_1@is_2,\ xt_1@shift\ (size\ is_1)\ xt_2\ [::]\ \tau\#\tau s_1@\tau'\#\tau s_2$

corollary (**in** *TC2*) *wt-instrs-Cons*[*simp*]:
  $[\![\tau s \neq []; \vdash [i],[] \; [::] \; [\tau,\tau']; \vdash is,xt \; [::] \; \tau'\#\tau s]\!]$
  $\Longrightarrow \vdash i\#is,shift\ 1\ xt \; [::] \; \tau\#\tau'\#\tau s$

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.