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

Gerwin Klein               Tobias Nipkow

September 19, 2015
## Contents

1 Preface
   1.1 Theory Dependencies ........................................ 5
   1.2 A table-based implementation of the reflexive transitive closure .... 6

2 Jinja Source Language ............................................. 11
   2.1 Auxiliary Definitions ........................................ 11
   2.2 Jinja types .................................................... 12
   2.3 Class Declarations and Programs .................................. 13
   2.4 Relations between Jinja Types .................................... 14
   2.5 Jinja Values ................................................... 20
   2.6 Objects and the Heap .......................................... 20
   2.7 Exceptions ..................................................... 22
   2.8 Expressions .................................................... 24
   2.9 Program State ................................................ 25
   2.10 Big Step Semantics ........................................... 26
   2.11 Small Step Semantics ......................................... 30
   2.12 System Classes ............................................... 34
   2.13 Generic Well-formedness of programs .......................... 35
   2.14 Weak well-formedness of Jinja programs ....................... 38
   2.15 Equivalence of Big Step and Small Step Semantics ............. 38
   2.16 Well-typedness of Jinja expressions........................... 44
   2.17 Runtime Well-typedness ....................................... 46
   2.18 Definite assignment ........................................... 48
   2.19 Conformance Relations for Type Soundness Proofs ............... 50
   2.20 Progress of Small Step Semantics .............................. 52
   2.21 Well-formedness Constraints ................................... 53
   2.22 Type Safety Proof ............................................ 54
   2.23 Program annotation .......................................... 56
   2.24 Example Expressions .......................................... 57
   2.25 Code Generation For BigStep .................................. 60
   2.26 Code Generation For WellType ................................. 63

3 Jinja Virtual Machine .............................................. 67
   3.1 State of the JVM ............................................... 67
   3.2 Instructions of the JVM ....................................... 67
   3.3 JVM Instruction Semantics .................................... 68
   3.4 Exception handling in the JVM ................................ 71
### 3.5 Program Execution in the JVM .................................................. 72
### 3.6 A Defensive JVM ...................................................................... 73
### 3.7 Example for generating executable code from JVM semantics ....... 76

#### 4 Bytecode Verifier

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Semilattices</td>
<td>81</td>
</tr>
<tr>
<td>4.2</td>
<td>The Error Type</td>
<td>84</td>
</tr>
<tr>
<td>4.3</td>
<td>More about Options</td>
<td>87</td>
</tr>
<tr>
<td>4.4</td>
<td>Products as Semilattices</td>
<td>87</td>
</tr>
<tr>
<td>4.5</td>
<td>Fixed Length Lists</td>
<td>88</td>
</tr>
<tr>
<td>4.6</td>
<td>Typing and Dataflow Analysis Framework</td>
<td>91</td>
</tr>
<tr>
<td>4.7</td>
<td>More on Semilattices</td>
<td>92</td>
</tr>
<tr>
<td>4.8</td>
<td>Lifting the Typing Framework to err, app, and eff</td>
<td>93</td>
</tr>
<tr>
<td>4.9</td>
<td>Kildall’s Algorithm</td>
<td>95</td>
</tr>
<tr>
<td>4.10</td>
<td>The Lightweight Bytecode Verifier</td>
<td>97</td>
</tr>
<tr>
<td>4.11</td>
<td>Correctness of the LBV</td>
<td>101</td>
</tr>
<tr>
<td>4.12</td>
<td>Completeness of the LBV</td>
<td>102</td>
</tr>
<tr>
<td>4.13</td>
<td>The Jinja Type System as a Semilattice</td>
<td>103</td>
</tr>
<tr>
<td>4.14</td>
<td>The JVM Type System as Semilattice</td>
<td>105</td>
</tr>
<tr>
<td>4.15</td>
<td>Effect of Instructions on the State Type</td>
<td>108</td>
</tr>
<tr>
<td>4.16</td>
<td>Monotonicity of eff and app</td>
<td>114</td>
</tr>
<tr>
<td>4.17</td>
<td>The Bytecode Verifier</td>
<td>114</td>
</tr>
<tr>
<td>4.18</td>
<td>The Typing Framework for the JVM</td>
<td>116</td>
</tr>
<tr>
<td>4.19</td>
<td>Kildall for the JVM</td>
<td>117</td>
</tr>
<tr>
<td>4.20</td>
<td>LBV for the JVM</td>
<td>119</td>
</tr>
<tr>
<td>4.21</td>
<td>BV Type Safety Invariant</td>
<td>120</td>
</tr>
<tr>
<td>4.22</td>
<td>BV Type Safety Proof</td>
<td>122</td>
</tr>
<tr>
<td>4.23</td>
<td>Welltyped Programs produce no Type Errors</td>
<td>127</td>
</tr>
<tr>
<td>4.24</td>
<td>Example Welltyprings</td>
<td>129</td>
</tr>
</tbody>
</table>

#### 5 Compilation

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>An Intermediate Language</td>
<td>137</td>
</tr>
<tr>
<td>5.2</td>
<td>Well-Formedness of Intermediate Language</td>
<td>141</td>
</tr>
<tr>
<td>5.3</td>
<td>Program Compilation</td>
<td>143</td>
</tr>
<tr>
<td>5.4</td>
<td>Compilation Stage 1</td>
<td>153</td>
</tr>
<tr>
<td>5.5</td>
<td>Correctness of Stage 1</td>
<td>154</td>
</tr>
<tr>
<td>5.6</td>
<td>Compilation Stage 2</td>
<td>156</td>
</tr>
<tr>
<td>5.7</td>
<td>Correctness of Stage 2</td>
<td>158</td>
</tr>
<tr>
<td>5.8</td>
<td>Combining Stages 1 and 2</td>
<td>162</td>
</tr>
<tr>
<td>5.9</td>
<td>Preservation of Well-Typedness</td>
<td>162</td>
</tr>
</tbody>
</table>
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.
1.2 A table-based implementation of the reflexive transitive closure

theory Transitive-Closure-Table
imports Main
begin

inductive rtrancl-path :: ('a ⇒ 'a ⇒ bool) ⇒ 'a ⇒ 'a list ⇒ '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: ∀ x y. (r∗∗ x y) ←→ (∃ xs. rtrancl-path r x xs y)
  proof
  show (∃ xs. rtrancl-path r x xs y) if r∗∗ x y
    using that
    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
  show r∗∗ x y if (∃ xs. rtrancl-path r x xs y)
    proof
      from that obtain xs where rtrancl-path r x xs y ..
      then show ?thesis
      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 zs)
then have \( r\text{trancl-path} r\ y\ (xs \@\ ys)\ z \)
  by simp
with \((r\ x\ y)\) have \( r\text{trancl-path} r\ x\ (y \# (xs \@\ ys))\ z \)
  by (rule \( r\text{trancl-path}.\text{step} \))
then show \(?\text{case}\) by simp
qed

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

lemma \( r\text{trancl-path-distinct} \):
assumes \( xy: r\text{trancl-path}\ r\ x\ xs\ y \)
obtains \( xs'\) where \( r\text{trancl-path}\ r\ x\ xs'\ y\) and \( \text{distinct}\ (x\#xs') \)
using \( xy \)
proof (induct \( xs \) rule: measure-induct-rule [of length])
case (less \( xs \))
  show \(?\text{case}\)
proof (cases \( \text{distinct}\ (x\#xs) \))
  case True
  with \( r\text{trancl-path}\ r\ x\ xs\ y \) show \(?\text{thesis}\)
    by (rule less)
next
  case False
  then have \( \exists as\ bs\ cs\ a.\ x\#xs = as\@[a]\@bs\@[a]\@cs \)
    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 xs x
  have x: x = a and xs: xs = bs @ a # cs
  by auto
from x x
  have cs: rtrancl-path r x cs y
  by (auto elim: rtrancl-path-appendE)
from x x
  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 xs x
  have x: x = 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 (rule rtrancl-path-trans)
from xa
  have xy: rtrancl-path r x ((ds @ [a]) @ cs) y
  by (rule rtrancl-path-trans)
from x x
  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
  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
  show rtrancl-tab r [] x y if r** x y
  proof
    from that 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 ?thesis
    by (rule rtrancl-path-imp-rtrancl-tab)
  qed
  show r** x y if rtrancl-tab r [] x y
  proof
    from that obtain xs where rtrancl-path r x xs y
    by (rule rtrancl-tab-imp-rtrancl-path)
    then show ?thesis
    by (auto simp add: rtranclp-eq-rtrancl-path)
  qed
qed

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

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

end
Figure 1.1: Theory Dependency Graph
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 ≤ m) = (n + i ≤ m ∧ n + j ≤ m)

lemma Suc-add-max-le[simp]:
(Suc(n + max i j) ≤ m) = (Suc(n + i) ≤ m ∧ Suc(n + j) ≤ m)

notation Some ((([-])))

2.1.1 distinct-fst

definition distinct-fst :: ('a × 'b) list ⇒ bool
where
  distinct-fst ≡ distinct o map fst

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

lemma distinct-fst-Cons [simp]:
distinct-fst ((k,x)#kxs) = (distinct-fst kxs ∧ (∀y. (k,y) /∈ set kxs))

lemma map-of-SomeI:
[ distinct-fst kxs; (k,x) ∈ set kxs ] ⇒ map-of kxs k = Some x

2.1.2 Using list-all2 for relations

definition fun-of :: ('a × 'b) set ⇒ 'a ⇒ 'b ⇒ bool
where
  fun-of S ≡ λx y. (x,y) ∈ S

  Convenience lemmas

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

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

lemma rel-list-all2-refl:
(∀ x. (x,x) ∈ S) ⇒ list-all2 (fun-of S) xs xs

lemma rel-list-all2-antisym:
(∀ x y. [(x,y) ∈ S; (y,x) ∈ T] ⇒ x = y);
list-all2 (fun-of S) xs ys; list-all2 (fun-of T) ys xs] ⇒ xs = ys

lemma rel-list-all2-trans:
(∀ a b c. [(a,b) ∈ R; (b,c) ∈ S] ⇒ (a,c) ∈ T);
list-all2 (fun-of R) as bs; list-all2 (fun-of S) bs cs]
⇒ list-all2 (fun-of T) as cs

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

lemma rel-list-all2-nthD:
(∀ list-all2 (fun-of S) xs ys; p < size xs ] ⇒ (xs[p],ys[p]) ∈ S

lemma rel-list-all2I:
(∀ length a = length b; ∀ n. n < length a ⇒ (a[n],b[n]) ∈ S ] ⇒ list-all2 (fun-of S) a b

end

2.2 Jinja types

theory Type imports Auxiliary begin

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

definition Object :: cname
where
Object ≡ "Object"

definition this :: vname
where
this ≡ "this"

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

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

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

2.3 Class Declarations and Programs

theory Decl imports Type begin

type-synonym fdecl = vname × ty — field declaration
type-synonym ′m mdecl = mname × ty list × ty × ′m — method = name, arg. types, return type, body
type-synonym ′m class = cname × fdecl list × ′m mdecl list — class = superclass, fields, methods
type-synonym ′m cdecl = cname × ′m class — class declaration
type-synonym ′m prog = ′m cdecl list — program
definition class :: ′m prog ⇒ cname → ′m class
where
class ≡ map-of
definition is-class :: ′m prog ⇒ cname ⇒ bool
where
  is-class P C ≡ class P C ≠ None

lemma finite-is-class: finite { C. is-class P C}
definition is-type :: ′m prog ⇒ ty ⇒ bool
where
  is-type P T ≡
  (case T of Void ⇒ True | Boolean ⇒ True | Integer ⇒ True | NT ⇒ True
   | Class C ⇒ is-class P C)

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


2.4 Relations between Jinja Types

theory TypeRel imports
  ~~/src/HOL/Library/Transitive-Closure-Table
begin

2.4.1 The subclass relations

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

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

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

2.4.2 The subtype relations

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

abbreviation (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 |- T ≤ Void) = (T = Void)
lemma [iff]: (P |- T ≤ Boolean) = (T = Boolean)
lemma [iff]: (P |- T ≤ Integer) = (T = Integer)
lemma [iff]: (P |- Void ≤ T) = (T = Void)
lemma [iff]: (P |- Boolean ≤ T) = (T = Boolean)
lemma [iff]: (P |- Integer ≤ T) = (T = Integer)

lemma Class-widen: P |- Class C ≤ T ⇒ ∃ D. T = Class D
2.4.3 Method lookup

inductive
Methods :: ['m prog, cname, mname -> (ty list x ty x 'm) x cname] => bool
(- |- sees-methods - [51,51,51] 50)
for P :: 'm prog

where
sees-methods-Object:
[ class P Object = Some(D,fs,ms); Mm = map-option (\m. (m,Object)) o map-of ms ]
=> P |- Object sees-methods Mm

| sees-methods-rec:
[ class P C = Some(D,fs,ms); C != Object; P |- D sees-methods Mm;
Mm' = Mm ++ (map-option (\m. (m,C)) o map-of ms) ]
=> P |- C sees-methods Mm'

lemma sees-methods-fun:
assumes 1: P |- C sees-methods Mm
shows \\[ Mm'. P |- C sees-methods Mm' \implies Mm' = Mm \]

lemma visible-methods-exist:
P |- C sees-methods Mm \implies Mm M = Some(m,D) \implies
(\exists D' fs ms. class P D = Some(D',fs,ms) \land map-of ms M = Some m)

lemma sees-methods-decl-above:
assumes Csees: P |- C sees-methods Mm
shows Mm M = Some(m,D) => P |- C <=* D

lemma sees-methods-idemp:
assumes Cmethods: P |- C sees-methods Mm
shows \\[ \forall m. Mm M = Some(m,D) \implies
\exists Mm'. (P |- D sees-methods Mm') \land Mm' M = Some(m,D) \]

lemma sees-methods-decl-mono:
assumes sub: P |- C' <=* C
shows P |- C sees-methods Mm =>
\exists Mm'. Mm2. P |- C' sees-methods Mm' \land Mm' = Mm ++ Mm2 \land
(\forall M m D. Mm2 M = Some(m,D) => P |- D <=* C)

definition Method :: 'm prog => cname => mname => ty list => ty => 'm => cname => bool
where
P |- C sees M: Ts => T = m in D \equiv
\exists Mm. P |- C sees-methods Mm \land Mm M = Some((Ts,T,m),D)

definition has-method :: 'm prog => cname => mname => bool (- |- has - [51,0,51] 50)
where
\[ P \vdash C \text{ has } M \equiv \exists T \ s. T \ m \ D. \ P \vdash C \text{ sees } M:Ts\rightarrow T = m \text{ in } D \]

**Lemma sees-method-fun:**
\[ [ P \vdash C \text{ sees } M:Ts\rightarrow T = m \text{ in } D; P \vdash C \text{ sees } M:Ts'\rightarrow T' = m' \text{ in } D' ] \]
\[ \implies TS' = TS \land T' = T \land m = m \land D = D \]

**Lemma sees-method-decl-above:**
\[ P \vdash C \text{ sees } M: Ts \rightarrow T = m \text{ in } D \implies P \vdash C \preceq^* D \]

**Lemma visible-method-exists:**
\[ P \vdash C \text{ sees } M: Ts \rightarrow T = m \text{ in } D \]
\[ \implies \exists D' \ s. ms. \class P D = \text{Some}(D',fs,ms) \land \text{map-of } ms M = \text{Some}(Ts,T,m) \]

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

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

### 2.4.4 Field lookup

**Inductive Fields**
\[ Fields ::= \prime \ m \ prog, \ curname, ((\prime tname \times \ curname) \times \prime ty) \ list \Rightarrow \text{bool} \]
\[ (- \vdash \prime has-fields - [51,51,51] 50) \]

**For P ::= \prime m \ prog**

**Where**

**has-fields-rec:**
\[ [ \class P C = \text{Some}(D,fs,ms); C \neq \text{Object}; P \vdash D \text{ has-fields } FDTs; \]
\[ FDTs' = \text{map } (\lambda(F,T). (\langle F,C,T \rangle)) \ fs \ominus FDTs ] \]
\[ \implies P \vdash C \text{ has-fields } FDTs' \]

**has-fields-Object:**
\[ [ \class P \text{Object} = \text{Some}(D,fs,ms); FDTs = \text{map } (\lambda(F,T). (\langle F,\text{Object},T \rangle)) \ fs ] \]
\[ \implies P \vdash \text{Object has-fields } FDTs \]

**Lemma has-fields-fun:**

**Assumes 1:**
\[ P \vdash C \text{ has-fields } FDTs \]

**Shows**
\[ \forall FDTs'. \ P \vdash C' \text{ has-fields } FDTs' \implies FDTs' = FDTs \]

**Lemma all-fields-in-has-fields:**

**Assumes sub:**
\[ P \vdash C \text{ has-fields } FDTs \]

**Shows**
\[ [ P \vdash C \preceq^* D; \class P D = \text{Some}(D',fs,ms); (F,T) \in \text{set } fs ] \]
\[ \implies ((F,D),T) \in \text{set } FDTs \]

**Lemma has-fields-decl-above:**

**Assumes fields:**
\[ P \vdash C \text{ has-fields } FDTs \]

**Shows**
\[ ((F,D),T) \in \text{set } FDTs \implies P \vdash C \preceq^* D \]

**Lemma subcls-notin-has-fields:**

**Assumes fields:**
\[ P \vdash C \text{ has-fields } FDTs \]
shows \((F,D), T) \in \text{set FDTs} \implies (D,C) \notin (\text{subcls} P)^+

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

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

\text{lemma has-field-mono:}
\Leftarrow \text{P } \vdash C \text{ has } F:T \text{ in } D
\exists \text{FDTs}. P \vdash C \text{ has-fields FDTs } \land \text{map-of FDTs } (F,D) = \text{Some } T

\text{definition sees-field :: 'm prog } \Rightarrow \text{cname } \Rightarrow \text{vname } \Rightarrow \text{ty } \Rightarrow \text{cname } \Rightarrow \text{bool}
\text{where}
P \vdash C \text{ sees } F:T \text{ in } D \equiv
\exists \text{FDTs}. P \vdash C \text{ sees } F:T \text{ in } D

\text{lemma map-of-remap-SomeD:}
\text{map-of } (\text{map } (\lambda((F,D),T). (F,(D,T))) \text{ FDTs}) F = \text{Some}(D,T)

\text{lemma has-visible-field:}
P \vdash C \text{ sees } F:T \text{ in } D \implies P \vdash C \text{ has } F:T \text{ in } D

\text{lemma sees-field-fun:}
\Leftarrow \text{P } \vdash C \text{ sees } F:T \text{ in } D \equiv
\exists \text{FDTs}. P \vdash C \text{ sees } F:T \text{ in } D

\text{lemma sees-field-decl-above:}
P \vdash C \text{ sees } F:T \text{ in } D \implies P \vdash C \le^* D

\text{lemma sees-field-idemp:}
P \vdash C \text{ sees } F:T \text{ in } D \implies P \vdash D \text{ sees } F:T \text{ in } D

\subsection*{2.4.5 Functional lookup}

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

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

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

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

## 2.4.6 Code generator setup

code-pred
\[(\text{modes}: i \Rightarrow i \Rightarrow i \Rightarrow \text{bool}, \ i \Rightarrow i \Rightarrow o \Rightarrow \text{bool})\]

\text{subcls1p}

\begin{itemize}
  \item declare \text{subcls1-def} \text{[code-pred-def]}
\end{itemize}

code-pred
\[(\text{modes}: i \Rightarrow i \times o \Rightarrow \text{bool}, \ i \Rightarrow i \Rightarrow o \Rightarrow \text{bool})\]

\text{[inductify]}

\text{subcls1}

\begin{itemize}
  \item definition \text{subcls'} where \text{subcls'} G = \text{(subcls1p G)}^{	ext{**}}
\end{itemize}

code-pred
\[(\text{modes}: i \Rightarrow i \Rightarrow i \Rightarrow \text{bool}, \ i \Rightarrow i \Rightarrow o \Rightarrow \text{bool})\]

\text{[inductify]}

\text{subcls'}

\begin{itemize}
  \item lemma \text{subcls-conv-subcls'} \text{[code-unfold]}:
  \[
  (\text{subcls1 G})^* = \{(C, D), \ \text{subcls'} G C D\}
  \]
  \[\text{by} (\text{simp add: subcls'}-def subcls1-def rtrancl-def)\]
\end{itemize}

code-pred
\[(\text{modes}: i \Rightarrow i \Rightarrow i \Rightarrow \text{bool})\]

\text{widen}

\begin{itemize}
  \item code-pred
  \[(\text{modes}: i \Rightarrow i \Rightarrow o \Rightarrow \text{bool})\]
  \text{Fields}
\end{itemize}

\begin{itemize}
  \item lemma \text{has-field-code} \text{[code-pred-intro]}:
  \[
  \begin{array}{ll}
  \text{[} & P \vdash C \text{has-fields FDTs} \\
  \text{map-of} & \text{FDTs} \ (F, D) = [T] \ \\
  \text{] } & P \vdash C \text{has } F : T \text{ in } D
  \end{array}
  \]
  \[\text{by}(\text{auto simp add: has-field-def})\]
\end{itemize}

code-pred
\[(\text{modes}: i \Rightarrow i \Rightarrow o \Rightarrow \text{bool}, \ i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow \text{bool})\]

\text{has-field}

\[\text{by}(\text{auto simp add: has-field-def})\]

\begin{itemize}
  \item lemma \text{sees-field-code} \text{[code-pred-intro]}:
  \[
  \begin{array}{ll}
  \text{[} & P \vdash C \text{has-fields FDTs} \\
  \text{map-of} & \text{(map } (\lambda((F, D), T). \ (F, D, T)) \text{ FDTs}) \ F = [(D, T)] \ \\
  \text{]} & P \vdash C \text{ sees } F : T \text{ in } D
  \end{array}
  \]
  \[\text{by}(\text{auto simp add: sees-field-def})\]
\end{itemize}

code-pred
\[(\text{modes}: i \Rightarrow i \Rightarrow o \Rightarrow o \Rightarrow \text{bool}, \ i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow i \Rightarrow \text{bool}, \ i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow \text{bool}, \ i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow \text{bool})\]
Theory TypeRel

sees-field
by(auto simp add: sees-field-def)

code-pred
(modes: \text{i} \Rightarrow \text{i} \Rightarrow \text{o} \Rightarrow \text{bool})
Methods
.

\text{lemma Method-code [code-pred-intro]:}
\begin{align*}
\ &\begin{array}{l}
P \vdash C \text{ sees-methods } Mm; \ Mm M = [\langle Ts, T, m \rangle, D]\end{array} \\
\Rightarrow P \vdash C \text{ sees } M : Ts \Rightarrow T = m \text{ in } D
\end{align*}
by(auto simp add: Method-def)

code-pred
(modes: \text{i} \Rightarrow \text{i} \Rightarrow \text{i} \Rightarrow \text{bool},
\text{i} \Rightarrow \text{i} \Rightarrow \text{i} \Rightarrow \text{i} \Rightarrow \text{i} \Rightarrow \text{i} \Rightarrow \text{bool})
Method
by(auto simp add: Method-def)

\text{lemma eval-Method-i-i-i-o-o-o-o-conv:}
\begin{align*}
\ &\begin{array}{l}
\text{Predicate.eval (Method-i-i-i-o-o-o-o P C M) = (}\lambda(Ts, T, m, D). \ P \vdash C \text{ sees } M : Ts \Rightarrow T = m \text{ in } D\end{array}
\end{align*}
by(auto intro: Method-i-i-i-o-o-o-oI elim: Method-i-i-i-o-o-o-oE intro: ext)

\text{lemma method-code [code]:}
\begin{align*}
\ &\begin{array}{l}
\text{method } P C M = \\
\text{Predicate.the (Predicate.bind (Method-i-i-i-o-o-o-o P C M) (}\lambda(Ts, T, m, D). \ 
\text{Predicate.single (D, Ts, T, m))))}
\end{array}
\end{align*}
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 \text{f = The}])
apply (auto simp add: SUP-def Sup-fun-def Sup-bool-def fun-eq-iff)
done

\text{lemma eval-Fields-conv:}
\begin{align*}
\ &\begin{array}{l}
\text{Predicate.eval (Fields-i-i-o P C) = (}\lambda FDTs. \ P \vdash C \text{ has-fields } FDTs\end{array}
\end{align*}
by(auto intro: Fields-i-i-oI elim: Fields-i-i-oE intro!: ext)

\text{lemma fields-code [code]:}
\begin{align*}
\ &\begin{array}{l}
\text{fields } P C = \text{Predicate.the (Fields-i-i-o P C)}
\end{array}
\end{align*}
by(simp add: fields-def Predicate.the-def eval-Fields-conv)

\text{lemma eval-sees-field-i-i-i-o-o-conv:}
\begin{align*}
\ &\begin{array}{l}
\text{Predicate.eval (sees-field-i-i-i-o-o P C F) = (}\lambda(T, D). \ P \vdash C \text{ sees } F : T \text{ in } D\end{array}
\end{align*}
by(auto intro!: ext intro: sees-field-i-i-i-o-oI elim: sees-field-i-i-i-o-oE)

\text{lemma eval-sees-field-i-i-i-o-o-conv:}
\begin{align*}
\ &\begin{array}{l}
\text{Predicate.eval (sees-field-i-i-i-o-o P C F D) = (}\lambda T. \ P \vdash C \text{ sees } F : T \text{ in } D\end{array}
\end{align*}
by(auto intro!: ext intro: sees-field-i-i-i-o-oI elim: sees-field-i-i-i-o-oE)

\text{lemma field-code [code]:}
\begin{align*}
\ &\begin{array}{l}
\text{field } P C F = \text{Predicate.the (Predicate.bind (sees-field-i-i-i-o-o P C F) (}\lambda(T, D). \ 
\text{Predicate.single (D, T)))}
\end{array}
\end{align*}
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 \( f = \text{The} \)])
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 o map (λ(F,T). (F,default-val T))

    — a new, blank object with default values in all fields:
Theory Objects

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-Addr h = Some a ⇒ h a = None

lemma [simp]: (typeof_h v = Some Boolean) = (∃ b. v = Bool b)

lemma [simp]: (typeof_h v = Some Integer) = (∃ i. v = Intg i)

lemma [simp]: (typeof_h v = Some NT) = (v = Null)

lemma [simp]: (typeof_h v = Some(Class C)) = (∃ a fs. v = Addr a ∧ h a = Some(C,fs))

lemma [simp]: h a = Some(C,fs) ⇒ typeof_h (h(a→(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 ⇒ ty option where
typeof v == typeof-h empty v

lemma typeof-lit-typeof:
typeof \( v \) = \text{Some} \ T \implies \text{typeof} \ h \ v = \text{Some} \ T

\text{lemma typeof-lit-is-type:}
\quad \text{typeof} \ v = \text{Some} \ T \implies \text{is-type} \ P \ T

### 2.6.3 Heap extension \( \preceq \)

\text{lemma hextI:} \quad \forall \ a \ C \ fs. \ h \ a = \text{Some} (C, fs) \implies \exists h', a = \text{Some} (C, fs') \implies h \preceq h'

\text{lemma hext-objD:} \quad \lbrack h \preceq h'; h \ a = \text{Some} (C, fs) \rbrack \implies \exists fs', h' \ a = \text{Some} (C, fs')

\text{lemma hext-refl [iff]:} \quad h \ h = \text{Some}(C, fs) = \exists fs' . h' = \text{Some}(C, fs')

\text{lemma hext-new [simp]:} \quad h \ a = \text{None} \implies h \preceq h(a \mapsto x)

\text{lemma hext-trans:} \quad \lbrack h \preceq h'; h' \preceq h'' \rbrack \implies h \preceq h''

\text{lemma hext-typeof-mono:} \quad \lbrack h \preceq h'; \text{typeof} h \ v = \text{Some} T \rbrack \implies \text{typeof} h' v = \text{Some} T

Code generator setup for \texttt{new-Addr}

\text{definition gen-new-Addr :: heap \Rightarrow addr \Rightarrow addr option}

\text{where gen-new-Addr h n \equiv if} \exists a . a \geq n \land h \ a = \text{None} \text{ then Some}(\text{LEAST} a . a \geq n \land h \ a = \text{None}) \text{ else None}

\text{lemma new-Addr-code-code [code]:}
\quad \text{new-Addr h = gen-new-Addr h 0}

\text{by(simp add: new-Addr-def gen-new-Addr-def split del: split-if cong: if-cong)}

\text{lemma gen-new-Addr-code [code]:}
\quad \text{gen-new-Addr h n = (if h n = None then Some n else gen-new-Addr h (Suc n))}

\text{apply(simp add: gen-new-Addr-def)}

\text{apply(rule impI)}

\text{apply(rule conjI)}

\text{apply safe[1]}

\text{apply(fastforce intro: Least-equality)}

\text{apply(rule arg-cong[where f=Least])}

\text{apply(rule ext)}

\text{apply(case-tac n = ac)}

\text{apply simp}

\text{apply(auto)[1]}

\text{apply clarify}

\text{apply(subgoal-tac a = n)}

\text{apply simp}

\text{apply(rule Least-equality)}

\text{apply auto[2]}

\text{apply(rule ccontr)}

\text{apply(erule-tac x = a in allE)}

\text{apply simp}

\text{done}

end

### 2.7 Exceptions

\text{theory Exceptions imports Objects begin}

\text{definition NullPointer :: cname}
Theory Exceptions

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 ∈ sys-xcpts ]] ⇒ cname-of h (addr-of-sys-xcpt C) = C

lemma typeof-ClassCast [simp]:
preallocated $h \Rightarrow \text{typeof}_h(\text{Addr}(\text{addr-of-sys-xcpt}\ \text{ClassCast})) = \text{Some}(\text{Class ClassCast})$

**Lemma**  \text{typeof-OutOfMemory} [simp]:

preallocated $h \Rightarrow \text{typeof}_h(\text{Addr}(\text{addr-of-sys-xcpt\OutOfMemory})) = \text{Some}(\text{Class OutOfMemory})$

**Lemma**  \text{typeof-NullPointer} [simp]:

preallocated $h \Rightarrow \text{typeof}_h(\text{Addr}(\text{addr-of-sys-xcpt NullPointer})) = \text{Some}(\text{Class NullPointer})$

**Lemma**  preallocated-hext:

$\prealloc h; h \leq h' \implies \prealloc h'$

**Lemma**  preallocated-start:

$\prealloc (\text{start-heap P})$

end

### 2.8 Expressions

**Theory** Expr

**Imports** ../Common/Exceptions

**Begin**

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

datatype $'a exp$

$= \text{new} \ 'cname$ — class instance creation

$\mid \text{Cast} \ 'cname \ ('a\ exp)$ — type cast

$\mid \text{Val} \ val$ — value

$\mid \text{BinOp} \ ('a\ exp) \ bop \ ('a\ exp)$ ($\leftarrow \leftrightarrow [80,0,81] 80$) — binary operation

$\mid \text{Var} \ 'a$ — local variable (incl. parameter)

$\mid \text{LAss} \ 'a \ ('a\ exp) \ (\leftarrow [90,90,90]$ — local assignment

$\mid \text{FAcc} \ ('a\ exp) \ vname \ 'cname \ (--\{\} [10,90,99,90] 90$ — field access

$\mid \text{FAss} \ ('a\ exp) \ vname \ 'cname \ ('a\ exp) \ (--\{\} : [10,90,99,90] 90$ — field assignment

$\mid \text{Call} \ ('a\ exp) \ mname \ ('a\ exp\ list) \ ('\cdot\{\} [90,99,0] 90$ — method call

$\mid \text{Block} \ 'a \ ty \ ('a\ exp) \ ('\{\cdot;\{\cdot\} \)

$\mid \text{Seq} \ ('a\ exp) \ ('a\ exp) \ (--\cdot [61,60] 60$ — statement

$\mid \text{Cond} \ ('a\ exp) \ ('a\ exp) \ ('a\ exp) \ (\text{if} \ ('\cdot\{\} \cdot\} else \ [80,79,79,79] 70$)

$\mid \text{While} \ ('a\ exp) \ ('a\ exp) \ ('\cdot\{\} [80,79,79] 70$)

$\mid \text{throw} \ ('a\ exp)\)

$\mid \text{TryCatch} \ ('a\ exp) \ 'cname \ 'a \ ('a\ exp) \ (\text{try} \cdot\text{catch} ('\cdot \cdot\cdot \cdot \cdot [0,99,80,79] 70)$

**Type Synonym**

$\text{expr} = vname \ exp$ — Jinja expression

**Type Synonym**

$J\text{-mb} = vname\ list \times \text{expr}$ — Jinja method body: parameter names and expression

**Type Synonym**

$J\text{-prog} = J\text{-mb} \ prog$ — Jinja program

The semantics of binary operators:

**Fun** binop :: $bop \times \text{val} \times \text{val} \Rightarrow \text{val\ option}$ where

$\text{binop}(\text{Eq}, v_1, v_2) = \text{Some}(\text{Bool}(v_1 = v_2))$

$\mid \text{binop}(\text{Add}, \text{Intg} \ i_1, \text{Intg} \ i_2) = \text{Some}(\text{Intg}(i_1 + i_2))$

$\mid \text{binop}(\text{bop}, v_1, v_2) = \text{None}$
lemma \([\text{simp}]:\)
\[
(\text{binop}(\text{Add}, v_1, v_2) = \text{Some } v) = (\exists i_1 i_2. v_1 = \text{Intg } i_1 \land v_2 = \text{Intg } i_2 \land v = \text{Intg}(i_1 + i_2))
\]

### 2.8.1 Syntactic sugar

**abbreviation** \((\text{input})\)
\[
\text{InitBlock} :: 'a \Rightarrow 'a \cdot e_1 e_2 == \{ V; T; V := e_1; e_2 \}
\]

**abbreviation** \((\text{unit})\)
\[
\text{unit} == \text{Val Unit}
\]

**abbreviation** \((\text{null})\)
\[
\text{null} == \text{Val Null}
\]

**abbreviation** \((\text{addr } a)\)
\[
\text{addr } a == \text{Val (Addr } a)\]

**abbreviation** \((\text{true})\)
\[
\text{true} == \text{Val (Bool True)}
\]

**abbreviation** \((\text{false})\)
\[
\text{false} == \text{Val (Bool False)}
\]

**abbreviation** \((\text{Throw})\)
\[
\text{Throw } a == \text{throw (Val (Addr } a)\}
\]

**abbreviation** \((\text{THROW})\)
\[
\text{THROW } xc == \text{Throw (addr-of-sys-xcpt } xc)\}
\]

### 2.8.2 Free Variables

**primrec** \(\text{fv} :: \text{expr } \Rightarrow \text{vname set}\) **and** \(\text{fvs} :: \text{expr list } \Rightarrow \text{vname set}\) **where**
\[
\text{fv} (\text{new } C) = \{\}
\]
\[
\text{fv} (\text{Cast } C e) = \text{fv } e
\]
\[
\text{fv} (\text{Val } v) = \{\}
\]
\[
\text{fv} (\text{Var } V) = \{V\}
\]
\[
\text{fv} (\text{LAss } V e) = \{V\} \cup \text{fv } e
\]
\[
\text{fv} (e \cdot \text{F} \{D\}) = \text{fv } e
\]
\[
\text{fv} (e \cdot \text{M}(es)) = \text{fv } e \cup \text{fvs es}
\]
\[
\text{fv} (\{V; T; e\}) = \text{fv } e - \{V\}
\]
\[
\text{fv} (e_1 ; e_2) = \text{fv } e_1 \cup \text{fv } e_2
\]
\[
\text{fv} (\text{if } (b) e_1 \text{ else } e_2) = \text{fv } b \cup \text{fv } e_1 \cup \text{fv } e_2
\]
\[
\text{fv} (\text{while } (b) e) = \text{fv } b \cup \text{fv } e
\]
\[
\text{fv} (\text{throw } e) = \text{fv } e
\]
\[
\text{fv} (\text{try } e_1 \text{ catch } (C V) e_2) = \text{fv } e_1 \cup (\text{fv } e_2 - \{V\})
\]
\[
\text{fvs} ([\]) = \{\}
\]
\[
\text{fvs} (e#es) = \text{fv } e \cup \text{fvs } es
\]

**lemma** \([\text{simp}]:\)
\[
\text{fvs} (es_1 @ es_2) = \text{fvs } es_1 \cup \text{fvs } es_2
\]

**lemma** \([\text{simp}]:\)
\[
\text{fvs} (\text{map Val } vs) = \{\}
\]

### 2.9 Program State

**theory** \(\text{State}\) **imports** ../Common/Exceptions begin

**type-synonym**
\[
\text{locals} = \text{vname } \rightarrow \text{val} \quad \text{— local vars, incl. params and “this”}
\]
type-synonym
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] 81)
and evals :: J-prog ⇒ expr list ⇒ state ⇒ expr list ⇒ state ⇒ bool
  (· ⊢ ((1⟨−/⟩) ⇒ (1⟨−/⟩)) [51,0,0,0] 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,s₀⟩ ⇒ ⟨addr a,(h,l)⟩; h a = Some(D,fs); P ⊢ D ≤* C ] ]
  =⇒ P ⊢ ⟨Cast C e,s₀⟩ ⇒ ⟨addr a,(h,l)⟩

| CastNull:
  P ⊢ ⟨e,s₀⟩ ⇒ ⟨null,s₁⟩ =⇒ P ⊢ ⟨Cast C e,s₀⟩ ⇒ ⟨null,s₁⟩

| CastFail:
  [ [ P ⊢ ⟨e,s₀⟩ ⇒ ⟨addr a,(h,l)⟩; h a = Some(D,fs); P ⊢ D ≤* C ] ]
  =⇒ P ⊢ ⟨Cast C e,s₀⟩ ⇒ ⟨THROW ClassCast,(h,l)⟩

| CastThrow:
  P ⊢ ⟨e,s₀⟩ ⇒ ⟨throw e′,s₁⟩ =⇒ P ⊢ ⟨Cast C e,s₀⟩ ⇒ ⟨throw e′,s₁⟩

| Val:
  P ⊢ ⟨Val v,s⟩ ⇒ ⟨Val v,s⟩

| BinOp:
  [ [ P ⊢ ⟨e₁,s₀⟩ ⇒ ⟨Val v₁,s₁⟩; P ⊢ ⟨e₂,s₁⟩ ⇒ ⟨Val v₂,s₂⟩; binop(bop,v₁,v₂) = Some v ] ]
  =⇒ P ⊢ ⟨e₁ <bop> e₂,s₀⟩ ⇒ ⟨Val v,s₂⟩
Theory BigStep

| BinOpThrow1: |
| P ⊢ ⟨e₁,s₀ ⟩ ⇒ ⟨throw e,s₁ ⟩  |
| P ⊢ ⟨e₁ <bop> e₂, s₀ ⟩ ⇒ ⟨throw e,s₁ ⟩  |

| BinOpThrow2: |
| [ P ⊢ ⟨e₁,s₀ ⟩ ⇒ ⟨Val v₁,s₁ ⟩; P ⊢ ⟨e₂,s₁ ⟩ ⇒ ⟨throw e,s₂ ⟩ ]  |
| ⇒ P ⊢ ⟨e₁ <bop> e₂,s₀ ⟩ ⇒ ⟨throw e,s₂ ⟩  |

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

| LAss: |
| [ P ⊢ ⟨e,s₀ ⟩ ⇒ ⟨Val v,(h,l)⟩; l' = l(V→v) ]  |
| ⇒ P ⊢ ⟨V:=e,s₀ ⟩ ⇒ ⟨unit,(h,l')⟩  |

| LAssThrow: |
| P ⊢ ⟨e,s₀ ⟩ ⇒ ⟨throw e',s₁ ⟩  |
| P ⊢ ⟨V:=e,s₀ ⟩ ⇒ ⟨throw e',s₁ ⟩  |

| FAss: |
| [ P ⊢ ⟨e,s₀ ⟩ ⇒ ⟨addr a,(h,l)⟩; h a = Some(C,fs); fs(F,D) = Some v ]  |
| ⇒ P ⊢ ⟨e·F(D),s₀ ⟩ ⇒ ⟨Val v,(h,l)⟩  |

| FAssNull: |
| P ⊢ ⟨e,s₀ ⟩ ⇒ ⟨null,s₁ ⟩  |
| P ⊢ ⟨e·F(D),s₀ ⟩ ⇒ ⟨THROW NullPointer,s₁ ⟩  |

| FAssThrow: |
| P ⊢ ⟨e,s₀ ⟩ ⇒ ⟨throw e',s₁ ⟩  |
| P ⊢ ⟨e·F(D),s₀ ⟩ ⇒ ⟨throw e',s₁ ⟩  |

| FAss: |
| [ P ⊢ ⟨e₁,s₀ ⟩ ⇒ ⟨addr a,s₁ ⟩; P ⊢ ⟨e₂,s₁ ⟩ ⇒ ⟨Val v,(h₂,l₂)⟩;  |
| h₂ a = Some(C,fs); fs' = fs((F,D)→v); h₂' = h₂(a→(C,fs' )) ]  |
| ⇒ P ⊢ ⟨e₁·F(D) := e₂,s₀ ⟩ ⇒ ⟨unit,(h₂',l₂)⟩  |

| FAssNull: |
| [ P ⊢ ⟨e₁,s₀ ⟩ ⇒ ⟨null,s₁ ⟩; P ⊢ ⟨e₂,s₁ ⟩ ⇒ ⟨Val v,s₂ ⟩ ]  |
| ⇒ P ⊢ ⟨e₁·F(D) := e₂,s₀ ⟩ ⇒ ⟨THROW NullPointer,s₂ ⟩  |

| FAssThrow1: |
| P ⊢ ⟨e₁,s₀ ⟩ ⇒ ⟨throw e',s₁ ⟩  |
| P ⊢ ⟨e₁·F(D) := e₂,s₀ ⟩ ⇒ ⟨throw e',s₁ ⟩  |

| FAssThrow2: |
| [ P ⊢ ⟨e₁,s₀ ⟩ ⇒ ⟨Val v,s₁ ⟩; P ⊢ ⟨e₂,s₁ ⟩ ⇒ ⟨throw e',s₂ ⟩ ]  |
| ⇒ P ⊢ ⟨e₁·F(D) := e₂,s₀ ⟩ ⇒ ⟨throw e',s₂ ⟩  |

| CallObjThrow: |
| P ⊢ ⟨e,s₀ ⟩ ⇒ ⟨throw e',s₁ ⟩  |
| P ⊢ ⟨e·M(ps),s₀ ⟩ ⇒ ⟨throw e',s₁ ⟩  |
\[
\begin{align*}
\text{CallParamsThrow:} & \\
\{ P \vdash (e, s_0) \Rightarrow (\text{Val } v, s_1); \ P \vdash (es, s_1) \Rightarrow (\text{map Val } v \circ \text{throw } ex \neq ex', s_2) \} \\
& \implies P \vdash e.M(es), s_0 \Rightarrow (\text{throw } ex, s_2) \\
\text{CallNull:} & \\
\{ P \vdash (e, s_0) \Rightarrow (\text{null}, s_1); \ P \vdash (ps, s_1) \Rightarrow (\text{map Val } vs, s_2) \} \\
& \implies P \vdash e.M(ps), s_0 \Rightarrow (\text{THROW NullPointer}, s_2) \\
\text{Call:} & \\
\{ P \vdash (e, s_0) \Rightarrow (\text{addr } a, s_1); \ P \vdash (ps, s_1) \Rightarrow (\text{map Val } vs, (h_2, l_2)) \} \\
& \quad \quad h_2 a = \text{Some}(C, fs); \ P \vdash C \text{ sees } M : Ts \rightarrow T = (\text{pns}, \text{body}) \text{ in } D; \\
& \quad \quad \text{length } vs = \text{length } pns; \ l_2' = [\text{this } \rightarrow \text{Addr } a, \ pns[\rightarrow] vs]; \\
& \quad \quad P \vdash (\text{body}, (h_2, l_2')) \Rightarrow (e', (h_3, l_3)) \} \\
& \implies P \vdash e.M(ps), s_0 \Rightarrow (e', (h_3, l_2)) \\
\text{Block:} & \\
& \quad P \vdash (e_0, (h_0, l_0(V := \text{None}))) \Rightarrow (e_1, (h_1, l_1)) \\
& \quad P \vdash \{(V : T, e_0, (h_0, l_0)) \Rightarrow (e_1, (h_1, l_1(V := h_0 V)) \} \\
\text{Seq:} & \\
\{ P \vdash (e_0, s_0) \Rightarrow (\text{Val } v, s_1); \ P \vdash (e_1, s_1) \Rightarrow (e_2, s_2) \} \\
& \implies P \vdash (e_0 ; e_1, s_0) \Rightarrow (e_2, s_2) \\
\text{SeqThrow:} & \\
& \quad P \vdash (e_0, s_0) \Rightarrow (\text{throw } e, s_1) \\
& \quad P \vdash (e_0 ; e_1, s_0) \Rightarrow (\text{throw } e, s_1) \\
\text{CondT:} & \\
\{ P \vdash (e, s_0) \Rightarrow (\text{true}, s_1); \ P \vdash (e_1, s_1) \Rightarrow (e', s_2) \} \\
& \implies P \vdash (\text{if } (e) e_1 \text{ else } e_2, s_0) \Rightarrow (e', s_2) \\
\text{CondF:} & \\
\{ P \vdash (e, s_0) \Rightarrow (\text{false}, s_1); \ P \vdash (e_2, s_1) \Rightarrow (e', s_2) \} \\
& \implies P \vdash (\text{if } (e) e_1 \text{ else } e_2, s_0) \Rightarrow (e', s_2) \\
\text{CondThrow:} & \\
& \quad P \vdash (e, s_0) \Rightarrow (\text{throw } e', s_1) \\
& \quad P \vdash (\text{if } (e) e_1 \text{ else } e_2, s_0) \Rightarrow (\text{throw } e', s_1) \\
\text{WhileF:} & \\
& \quad P \vdash (e, s_0) \Rightarrow (\text{false}, s_1) \\
& \quad P \vdash (\text{while } (e) c, s_0) \Rightarrow (\text{unit}, s_1) \\
\text{WhileT:} & \\
\{ P \vdash (e, s_0) \Rightarrow (\text{true}, s_1); \ P \vdash (c, s_1) \Rightarrow (\text{Val } v_1, s_2); \ P \vdash (\text{while } (e) c, s_2) \Rightarrow (e_3, s_3) \} \\
& \implies P \vdash (\text{while } (e) c, s_0) \Rightarrow (e_3, s_3) \\
\text{WhileCondThrow:} & \\
& \quad P \vdash (e, s_0) \Rightarrow (\text{throw } e', s_1) \\
& \quad P \vdash (\text{while } (e) c, s_0) \Rightarrow (\text{throw } e', s_1) \\
\text{WhileBodyThrow:} & \\
\{ P \vdash (e, s_0) \Rightarrow (\text{true}, s_1); \ P \vdash (e, s_1) \Rightarrow (\text{throw } e', s_2) \} \\
& \implies P \vdash (\text{while } (e) c, s_0) \Rightarrow (\text{throw } e', s_2)
\end{align*}
\]
Theory BigStep

\[ \text{final} :: \forall a. \exp \Rightarrow \text{bool} \]

where
\[
\text{final } e \equiv (\exists v. e = \text{Val} v) \lor (\exists a. e = \text{Throw} a)
\]

\[ \text{finals} :: \forall a. \exp \Rightarrow \text{bool} \]

where
\[
\text{finals } es \equiv (\exists s. es = \text{map} \text{ Val vs}) \lor (\exists vs a es'. es = \text{map} \text{ Val vs @ Throw} a \# es')
\]

lemma simp: final(\text{Val} v)
lemma simp: final(\text{throw} e) = (\exists a. e = \text{addr} a)
lemma finalE: \[ \text{final } e; \ \forall v. e = \text{Val} v \Rightarrow R; \ \forall a. e = \text{Throw} a \Rightarrow R \ \Rightarrow R \]
lemma iff: \text{finals} []
lemma iff: finals (\text{Val} v \# es) = finals es
lemma finals-app-map[iff]: finals (\text{map} \text{ Val vs @ es}) = finals es
lemmas \textbf{blocks-induct} = blocks.induct[split-format (complete)]

lemma \textbf{simp}:
\[
\begin{array}{l}
\quad \quad [\text{size } vs = \text{size } Vs; \text{size } Ts = \text{size } Vs ] \Rightarrow fe(blocks(Vs,Ts,vs,e)) = fv e - \text{set } Vs
\end{array}
\]
definition \textbf{assigned} :: \text{vname} \Rightarrow \text{expr} \Rightarrow \text{bool}
where
\quad \textbf{assigned} V e \equiv \exists v' e. e = (V := \text{Val } v'; e')

inductive-set
\begin{array}{l}
\textbf{red} :: \text{J-prog} \Rightarrow ((\text{expr } \times \text{ state}) \times (\text{expr } \times \text{ state})) \ \text{set}
\quad \text{and } \textbf{reds} :: \text{J-prog} \Rightarrow ((\text{expr list } \times \text{ state}) \times (\text{expr list } \times \text{ state})) \ \text{set}
\end{array}
\begin{array}{l}
\quad \textbf{red}' :: \text{J-prog} \Rightarrow \text{expr } \Rightarrow \text{state} \Rightarrow \text{expr } \Rightarrow \text{state} \Rightarrow \text{bool}
\end{array}
\begin{array}{l}
\quad \text{and } \textbf{reds}' :: \text{J-prog} \Rightarrow \text{expr list } \Rightarrow \text{state} \Rightarrow \text{expr list } \Rightarrow \text{state} \Rightarrow \text{bool}
\end{array}
\begin{array}{l}
\quad \text{for } P :: \text{J-prog}
\end{array}
\begin{array}{l}
\quad \text{where}
\end{array}
\begin{array}{l}
\quad P \vdash (e,s) \Rightarrow (e',s') \equiv ((e,s), e',s') \in \text{red } P
\end{array}
\begin{array}{l}
\quad | P \vdash (e,s) \Rightarrow (es',s') \equiv ((es,s), es',s') \in \text{reds } P
\end{array}
<table>
<thead>
<tr>
<th>RedNew:</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ \text{new-Addr } h = \text{Some } a; P \vdash C \text{ has-fields FDTs}; h' = h(a \rightarrow (C, \text{init-fields FDTs})) ]</td>
</tr>
<tr>
<td>[ \implies P \vdash \langle \text{new } C, (h,l) \rangle \rightarrow \langle \text{addr } a, (h',l) \rangle ]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RedNewFail:</th>
</tr>
</thead>
<tbody>
<tr>
<td>new-Addr ( h = \text{None} \implies P \vdash \langle \text{new } C, (h,l) \rangle \rightarrow \langle \text{THROW OutOfMemory}, (h,l) \rangle )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CastRed:</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ P \vdash (\text{Cast } C e, s) \rightarrow \langle \text{Cast } C e', s' \rangle ]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RedCastNull:</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ P \vdash \langle \text{Cast } C \text{ null}, s \rangle \rightarrow \langle \text{null}, s' \rangle ]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RedCast:</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ \text{hp } a = \text{Some}(D, f) \vdash D \subseteq^* C ]</td>
</tr>
<tr>
<td>[ \implies P \vdash \langle \text{Cast } C (\text{addr } a), s \rangle \rightarrow \langle \text{addr } a, s \rangle ]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RedCastFail:</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ \text{hp } a = \text{Some}(D, f) ; \neg P \vdash D \subseteq^* C ]</td>
</tr>
<tr>
<td>[ \implies P \vdash \langle \text{Cast } C (\text{addr } a), s \rangle \rightarrow \langle \text{THROW ClassCast}, s \rangle ]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>BinOpRed1:</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ P \vdash (e, s) \rightarrow (e', s') \implies P \vdash (e \left&lt; bop \right&gt; e_2, s) \rightarrow (e' \left&lt; bop \right&gt; e_2, s') ]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>BinOpRed2:</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ P \vdash (\langle \text{Val } v_1 \rangle \left&lt; bop \right&gt; e, s) \rightarrow (\langle \text{Val } v_1 \rangle \left&lt; bop \right&gt; e', s') ]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RedBinOp:</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ \text{binop}(bop, v_1, v_2) = \text{Some } v \implies ]</td>
</tr>
<tr>
<td>[ P \vdash (\langle \text{Val } v_1 \rangle \left&lt; bop \right&gt; (\text{Val } v_2), s) \rightarrow (\text{Val } v, s) ]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RedVar:</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ \text{let } s \ V = \text{Some } v \implies ]</td>
</tr>
<tr>
<td>[ P \vdash (\text{Var } V, s) \rightarrow (\text{Val } v, s) ]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>LAssRed:</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ P \vdash (e, s) \rightarrow (e', s') \implies ]</td>
</tr>
<tr>
<td>[ P \vdash (V := e, s) \rightarrow (V := e', s') ]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RedLAss:</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ P \vdash (V := (\text{Val } v), (h,l)) \rightarrow (\text{unit}, (h,l(V := v))) ]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FAccRed:</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ P \vdash (e, s) \rightarrow (e', s') \implies ]</td>
</tr>
<tr>
<td>[ P \vdash (e \cdot F(D), s) \rightarrow (e' \cdot F(D), s') ]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RedFAcc:</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ \text{hp } a = \text{Some}(C, f); \text{fs}(F, D) = \text{Some } v ]</td>
</tr>
<tr>
<td>[ \implies P \vdash (\langle \text{addr } a \rangle \cdot F(D), s) \rightarrow (\text{Val } v, s) ]</td>
</tr>
</tbody>
</table>
```
| RedFAccNull: & P ⊢ ⟨null·F[D], s⟩ → ⟨THROW NullPointer, s⟩ |
| RedFCallRed1: & P ⊢ ⟨e,s⟩ → ⟨e′,s′⟩ ⊢
                    & P ⊢ ⟨e·F[D]:=e₂, s⟩ → ⟨e′·F[D]:=e₂, s′⟩ |
| RedFCallRed2: & P ⊢ ⟨e,s⟩ → ⟨e′,s′⟩ ⊢
                    & P ⊢ ⟨Val v·F[D]:=e, s⟩ → ⟨Val v·F[D]:=e′, s′⟩ |
| RedFAss:       & h a = Some(C,fs) ⊢
                    & P ⊢ ⟨⟨addr a⟩·F[D]:=(Val v), (h,l)⟩ → ⟨unit, (h(a → (C,fs((F,D) ↦ v)))),l⟩⟩ |
| RedFAssNull:   & P ⊢ ⟨null·F[D]:=Val v, s⟩ → ⟨THROW NullPointer, s⟩ |
| CallObj:       & P ⊢ ⟨e,s⟩ → ⟨e′,s′⟩ ⊢
                    & P ⊢ ⟨e·M(es),s⟩ → ⟨e′·M(es),s′⟩ |
| CallParams:    & P ⊢ ⟨es,s⟩ → ⟨es′,s′⟩ ⊢
                    & P ⊢ ⟨(Val v)·M(es),s⟩ → ⟨(Val v)·M(es′),s′⟩ |
| RedCall:       & P ⊢ ⟨hp s a = Some(C,fs); P ⊢ C sees M:T→T = (pns,body) in D; size vs = size pns; size Ts = size pns |
                    & P ⊢ ⟨⟨addr a⟩·M(map Val vs), s⟩ → ⟨blocks(this#pns, Class D#Ts, Addr a#vs, body), s⟩⟩ |
| RedCallNull:   & P ⊢ ⟨null·M(map Val vs),s⟩ → ⟨THROW NullPointer,s⟩ |
| BlockRedNone:  & P ⊢ ⟨e, (h,l(V:=None))⟩ → ⟨e′, (h′,l′)]; l′ V = None; ¬ assigned V e ⟩
                    & P ⊢ ⟨⟨V:T; e⟩, (h,l)⟩ → ⟨⟨V:T; e′⟩, (h′,l′(V := l V))⟩ |
| BlockRedSome:  & P ⊢ ⟨e, (h,l(V:=None))⟩ → ⟨e′, (h′,l′)]; l′ V = Some v; ¬ assigned V e ⟩
                    & P ⊢ ⟨⟨V:T; e⟩, (h,l)⟩ → ⟨⟨V:T := Val v; e⟩, (h′,l′(V := l V))⟩ |
| InitBlockRed:  & P ⊢ ⟨e, (h,l(V:=v))⟩ → ⟨e′, (h′,l′)]; l′ V = Some v’ ⟩
                    & P ⊢ ⟨⟨V:T := Val v; e⟩, (h,l)⟩ → ⟨⟨V:T := Val v’; e⟩, (h′,l′(V := l V))⟩ |
| RedBlock:      & P ⊢ ⟨⟨V:T := Val v; Val u⟩, s⟩ → ⟨Val u, s⟩ |
| RedInitBlock:  & P ⊢ ⟨⟨V:T := Val v; Val u⟩, s⟩ → ⟨Val u, s⟩ |
| SeqRed:        & |
```
Theory SmallStep

\[ P \vdash (e, s) \rightarrow (e', s') \]
\[ P \vdash (e; e_2, s) \rightarrow (e'; e_2, s') \]

<table>
<thead>
<tr>
<th>RedSeq:</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ P \vdash ((\text{Val } v);; e_2, s) \rightarrow (e_2, s) ]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CondRed:</th>
</tr>
</thead>
</table>
| \[ P \vdash (e, s) \rightarrow (e', s') \]
| \[ P \vdash (\text{if } (e) \text{ e}_1 \text{ else } e_2, s) \rightarrow (\text{if } (e') \text{ e}_1 \text{ else } e_2, s') \] |

<table>
<thead>
<tr>
<th>RedCondT:</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ P \vdash (\text{if } (\text{true}) \text{ e}_1 \text{ else } e_2, s) \rightarrow (e_1, s) ]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RedCondF:</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ P \vdash (\text{if } (\text{false}) \text{ e}_1 \text{ else } e_2, s) \rightarrow (e_2, s) ]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RedWhile:</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ P \vdash (\text{while} (b) \text{ c}, s) \rightarrow (\text{if } (b) \text{ c};; \text{while} (b) \text{ c}) \text{ else } \text{unit}, s) ]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ThrowRed:</th>
</tr>
</thead>
</table>
| \[ P \vdash (e, s) \rightarrow (e', s') \]
| \[ P \vdash (\text{throw } e, s) \rightarrow (\text{throw } e', s') \] |

<table>
<thead>
<tr>
<th>RedThrowNull:</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ P \vdash (\text{throw null}, s) \rightarrow (\text{THROW NullPointer}, s) ]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TryRed:</th>
</tr>
</thead>
</table>
| \[ P \vdash (e, s) \rightarrow (e', s') \]
| \[ P \vdash (\text{try } e \text{ catch}(C \text{ V}) e_2, s) \rightarrow (\text{try } e' \text{ catch}(C \text{ V}) e_2, s') \] |

<table>
<thead>
<tr>
<th>RedTryCatch:</th>
</tr>
</thead>
</table>
| \[ [ [ \text{hp } s \text{ a } = \text{Some}(D, fs); \ P \vdash D \preceq^* C ] ] \]
| \[ \implies P \vdash (\text{try } (\text{Throw a}) \text{ catch}(C \text{ V}) e_2, s) \rightarrow (\{(V:\text{Class } := \text{addr } a; e_2\}, s) \] |

<table>
<thead>
<tr>
<th>RedTryFail:</th>
</tr>
</thead>
</table>
| \[ [ [ \text{hp } s \text{ a } = \text{Some}(D, fs); \neg P \vdash D \preceq^* C ] ] \]
| \[ \implies P \vdash (\text{try } (\text{Throw a}) \text{ catch}(C \text{ V}) e_2, s) \rightarrow (\text{Throw a}, s) \] |

<table>
<thead>
<tr>
<th>ListRed1:</th>
</tr>
</thead>
</table>
| \[ P \vdash (e, s) \rightarrow (e', s') \]
| \[ P \vdash (e \# es, s) \rightarrow [e'] \rightarrow (e' \# es, s') \] |

<table>
<thead>
<tr>
<th>ListRed2:</th>
</tr>
</thead>
</table>
| \[ P \vdash (es, s) \rightarrow [s] \rightarrow (es', s') \]
| \[ P \vdash (\text{Val } v \# es, s) \rightarrow [s] \rightarrow (\text{Val } v \# es', s') \] |

— Exception propagation

<table>
<thead>
<tr>
<th>CastThrow:</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ P \vdash (\text{Cast } C \text{ (throw } e), s) \rightarrow (\text{throw } e, s) ]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>BinOpThrow1:</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ P \vdash ((\text{throw } e) \text{ &lt;bop&gt; } e_2, s) \rightarrow (\text{throw } e, s) ]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>BinOpThrow2:</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ P \vdash ((\text{Val } v_1) \text{ &lt;bop&gt; } (\text{throw } e), s) \rightarrow (\text{throw } e, s) ]</td>
</tr>
</tbody>
</table>
2.11.1 The reflexive transitive closure

abbreviation

Step :: J-prog ⇒ expr ⇒ state ⇒ expr ⇒ state ⇒ bool

(¬(t((1,/-)) / (1,/-))) [51,0,0,0,0] 81

where P ⊢ (e,s) →* (e′,s′) ≡ ((e,s), e′,s′) ∈ (red P)*

abbreviation

Steps :: J-prog ⇒ expr list ⇒ state ⇒ expr list ⇒ state ⇒ bool

(¬((1,/-)) / (1,/-))) [51,0,0,0,0] 81

where P ⊢ (es,s) →* (es′,s′) ≡ (es,s), es′,s′) ∈ (red P)*

lemma converse-rtrancl-induct-red[consumes 1]:

assumes P ⊢ (e,h,l) →* (e′,h′,l′)

and fs e h l. R e h l e h l

and ∨e0 h0 l0 e1 h1 l1 e′ h′ l′.

[ P ⊢ (e0,h0,l0) → (e1,(h1,l1)); R e1 h1 l1 e′ h′ l′ ] → R e0 h0 l0 e′ h′ l′

shows R e h l e′ h′ l′

2.12 Some easy lemmas

lemma [iff]: ¬P ⊢ [[s],s] → [[e′,s′]

lemma [iff]: P ⊢ (Val v,s) → (e′,s′)

lemma [iff]: ¬P ⊢ (Throw a,s) → (e′,s′)

lemma red-hext-incr: P ⊢ (e,h,l) → (e′,h′,l′) " h ≤ h′

and reds-hext-incr: P ⊢ (es,h,l) →* (es′,h,l′) " h ≤ h′

lemma red-lcl-incr: P ⊢ (e,h,l0) → (e′,h',l1) " dom l0 ⊆ dom l1

and P ⊢ (es,h0,l0) →* (es′,h1,l1) " dom l0 ⊆ dom l1

lemma red-lcl-add: P ⊢ (e,h,l) → (e′,h,l′) " (λl0. P ⊢ (e,h,l0++l) → (e′,(h','l0++l'))

and P ⊢ (es,h,l) →* (es′,h,l′) " (λl0. P ⊢ (es,h,l0++l) →* (es′,h',l0++l'))

lemma Red-lcl-add: assumes P ⊢ (e,h,l) →* (e′,h',l′) shows P ⊢ (e,(h,l0++l)) →* (e′,(h',l0++l'))

end

2.12 System Classes

theory SystemClasses
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]

2.13 Generic Well-formedness of programs

theory WellForm imports Decl Exceptions 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 cdecl ⇒ 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:Tss → 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 ⊢ C ≺1 D; wf-prog wf-md P ] ⇒ D ≠ C ∧ (D,C) ∉ (subcls1 P)†

lemma wf-cdecl-supD:
[wf-cdecl wf-md P (C,D,r); C ≠ Object] ⇒ is-class P D

lemma subcls-asym:
[wf-prog wf-md P; (C,D) ∈ (subcls1 P)† ] ⇒ (D,C) ∉ (subcls1 P)†

lemma subcls-irrefl:
[wf-prog wf-md P; (C,D) ∈ (subcls1 P)† ] ⇒ C ≠ D

lemma acyclic-subcls1:
wf-prog wf-md P ⇒ acyclic (subcls1 P)

lemma wf-subcls1:
wf-prog wf-md P ⇒ wf ((subcls1 P)−1)

lemma single-valued-subcls1:
wf-prog wf-md G ⇒ single-valued (subcls1 G)

lemma subcls-induct:
[wf-prog wf-md P; (∀ C. ∀ D. (C,D) ∈ (subcls1 P)† → Q D ⇒ Q C ] ⇒ Q C

lemma subcls1-induct-aux:
[ is-class P C; wf-prog wf-md P; Q Object;
\[ \land D fs ms. \[
C \neq Object; is\text{-}class P C; class P C = Some (D,fs,ms) \land \]
wf-cdecl wf-md P (C,D,fs,ms) \land P \vdash C \preceq D \land is\text{-}class P D \land Q D \] \implies Q C \]

**Lemma** subcls1-induct [consumes 2, case-names Object Subcls]:
\[
\land C D. [C \neq Object; P \vdash C \preceq D; is\text{-}class P D; Q D] \implies Q C \]

**Lemma** subcls-C-Object:
\[
\land is\text{-}class P C; wf-prog wf-md P \] \implies P \vdash C \preceq* Object

**Assumes** wf-prog wf-md P and (C,S,fs,ms) \in set P and (M,Ts,T,m) \in set ms
**Shows** set Ts \subseteq types P

### 2.13.2 Well-formedness and method lookup

**Lemma** sees-wf-mdecl:
\[
\land wf-prog wf-md P; P \vdash C sees M:Ts\rightarrow T = m in D \] \implies wf-mdecl wf-md P D (M,Ts,T,m)

**Lemma** sees-method-mono [rule-format (na-asf)]:
\[
\land P \vdash C' \preceq* C; wf-prog wf-md P \] \implies 
\begin{align*}
\forall D Ts T m. P \vdash C sees M:Ts\rightarrow T = m in D \implies 
(\exists D' Ts' T' m'. P \vdash C' sees M:Ts'\rightarrow T' = m' in D' \land P \vdash Ts [\leq] Ts' \land P \vdash T' \leq T)
\end{align*}

**Lemma** sees-method-mono2:
\[
\land P \vdash C' \preceq* C; wf-prog wf-md P; P \vdash C sees M:Ts\rightarrow T = m in D; P \vdash C' sees M:Ts'\rightarrow T' = m' in D' \]
\implies P \vdash Ts [\leq] Ts' \land P \vdash T' \leq T

**Lemma** mdecls-visible:
**Assumes** wf: wf-prog wf-md P and class: is-class P C
**Shows** \land D fs ms. class P C = Some(D,fs,ms)
\[ \implies \exists Mm. P \vdash C sees-methods Mm \land (\forall (M,Ts,T,m) \in set ms. Mm M = Some((Ts,T,m),C)) \]

**Lemma** mdecl-visible:
**Assumes** wf: wf-prog wf-md P and C: (C,S,fs,ms) \in set P and m: (M,Ts,T,m) \in set ms
**Shows** P \vdash C sees M:Ts\rightarrow T = m in C

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

**Lemma** wf-prog-lift:
**Assumes** wf: wf-prog (\lambda P C bd. A P C bd) P
**And** rule:
\land wf-md C M Ts C T m bd.
\land wf-prog wf-md P \implies 
P \vdash C sees M:Ts\rightarrow T = m in C \implies 
set Ts \subseteq types P \implies
2.13.3 Well-formedness and field lookup

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

lemma \textit{has-fields-types}:
\[
\text{[ \text{P \vdash C sees F:T in D}; \text{wf-prog wf-md P} ]} \implies \text{is-type P T}
\]

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

lemma \textit{wf-syscls}:
\[
\text{set SystemClasses} \subseteq \text{set P} \implies \text{wf-syscls P}
\]

2.14 Weak well-formedness of Jinja programs

theory \textit{WWellForm} imports ..../Common/WellForm Expr begin

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

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

abbreviation \textit{wwf-J-prog} :: \textit{J-prog} \Rightarrow \textit{bool}
where
\[
\text{wwf-J-prog} \equiv \text{wf-prog wwf-J-mdecl}
\]

end

2.15 Equivalence of Big Step and Small Step Semantics

theory \textit{Equivalence} imports BigStep SmallStep WWellForm begin

2.15.1 Small steps simulate big step

Cast

lemma \textit{CastReds}:
\[
P \vdash \langle e,s \rangle \rightarrow^* \langle e',s' \rangle \implies P \vdash \langle \text{Cast C e},s \rangle \rightarrow^* \langle \text{Cast C e'},s' \rangle
\]

lemma \textit{CastRedsNull}:
\[
P \vdash \langle e,s \rangle \rightarrow^* \langle \text{null},s' \rangle \implies P \vdash \langle \text{Cast C e},s \rangle \rightarrow^* \langle \text{null},s' \rangle
\]

lemma \textit{CastRedsAddr}:
\[
\text{[ P \vdash \langle e,s \rangle \rightarrow^* \langle \text{addr a},s' \rangle; \text{hp s'} a = \text{Some(D,fs)}; P \vdash D \preceq^* C ] \implies}
\text{ P \vdash \langle \text{Cast C e},s \rangle \rightarrow^* \langle \text{addr a},s' \rangle}
\]

lemma \textit{CastRedsFail}:
\[
\text{[ P \vdash \langle e,s \rangle \rightarrow^* \langle \text{addr a},s' \rangle; \text{hp s'} a = \text{Some(D,fs)}; \lnot P \vdash D \preceq^* C ] \implies}
\]
\[ P \vdash (\text{Cast } C \ e, s) \Rightarrow (\text{THROW ClassCast}, s') \]

**lemma** \textbf{CastRedsThrow}: 
\[ \begin{array}{l}
P \vdash \langle e, s \rangle \Rightarrow (\text{throw } a, s') \quad \Rightarrow \quad P \vdash (\text{Cast } C \ e, s) \Rightarrow (\text{throw } a, s')
\end{array} \]

**LAss**

**lemma** \textbf{LAssReds}: 
\[ P \vdash \langle e, s \rangle \Rightarrow \langle e', s' \rangle \quad \Rightarrow \quad P \vdash (V := e, s) \Rightarrow (V := e', s') \]

**lemma** \textbf{LAssRedsVal}: 
\[ \begin{array}{l}
P \vdash \langle e, s \rangle \Rightarrow (\text{Val } v, (h', l')) \quad \Rightarrow \quad P \vdash (V := e, s) \Rightarrow (\text{unit }, (h', l'(V \Rightarrow v)))
\end{array} \]

**lemma** \textbf{LAssRedsThrow}: 
\[ \begin{array}{l}
P \vdash \langle e, s \rangle \Rightarrow (\text{throw } a, s') \quad \Rightarrow \quad P \vdash (V := e, s) \Rightarrow (\text{throw } a, s')
\end{array} \]

**BinOp**

**lemma** \textbf{BinOp1Reds}: 
\[ P \vdash \langle e, s \rangle \Rightarrow \langle e', s' \rangle \quad \Rightarrow \quad P \vdash (e \ \text{binop} \ e_2, s) \Rightarrow (e' \ \text{binop} \ e_2, s') \]

**lemma** \textbf{BinOp2Reds}: 
\[ P \vdash \langle e, s \rangle \Rightarrow \langle e', s' \rangle \quad \Rightarrow \quad P \vdash ((\text{Val } v) \ \text{binop} \ e, s) \Rightarrow ((\text{Val } v) \ \text{binop} \ e', s') \]

**lemma** \textbf{BinOpRedsVal}: 
\[ \begin{array}{l}
P \vdash \langle e_1, s_0 \rangle \Rightarrow (\text{Val } v_1, s_1); P \vdash \langle e_2, s_1 \rangle \Rightarrow (\text{Val } v_2, s_2); \text{ binop}(\text{binop}, v_1, 2) = \text{ Some } v
\end{array} \]
\[ \Rightarrow P \vdash \langle e_1 \ \text{binop} \ e_2, s_0 \rangle \Rightarrow (\text{Val } v, s_2) \]

**lemma** \textbf{BinOpRedsThrow1}: 
\[ P \vdash \langle e, s \rangle \Rightarrow (\text{throw } e', s') \quad \Rightarrow \quad P \vdash (e \ \text{binop} \ e_2, s) \Rightarrow (\text{throw } e', s') \]

**lemma** \textbf{BinOpRedsThrow2}: 
\[ \begin{array}{l}
P \vdash \langle e_1, s_0 \rangle \Rightarrow (\text{Val } v_1, s_1); P \vdash \langle e_2, s_1 \rangle \Rightarrow (\text{throw } e, s_2)
\end{array} \]
\[ \Rightarrow P \vdash \langle e_1 \ \text{binop} \ e_2, s_0 \rangle \Rightarrow (\text{throw } e, s_2) \]

**FAcc**

**lemma** \textbf{FAccReds}: 
\[ P \vdash \langle e, s \rangle \Rightarrow \langle e', s' \rangle \quad \Rightarrow \quad P \vdash (e \ \text{-} \ F\{D\}, s) \Rightarrow (e' \ F\{D\}, s') \]

**lemma** \textbf{FAccRedsVal}: 
\[ \begin{array}{l}
P \vdash \langle e, s \rangle \Rightarrow (\text{addr } a, s'); P \vdash (\text{hp } s' a = \text{ Some } (C, fs); fs(F, D) = \text{ Some } v)
\end{array} \]
\[ \Rightarrow P \vdash (e \ F\{D\}, s) \Rightarrow (\text{Val } v, s') \]

**lemma** \textbf{FAccRedsNull}: 
\[ P \vdash \langle e, s \rangle \Rightarrow (\text{null}, s') \quad \Rightarrow \quad P \vdash (e \ F\{D\}, s) \Rightarrow (\text{THROW NullPointer}, s') \]

**lemma** \textbf{FAccRedsThrow}: 
\[ P \vdash \langle e, s \rangle \Rightarrow (\text{throw } a, s') \quad \Rightarrow \quad P \vdash (e \ F\{D\}, s) \Rightarrow (\text{throw } a, s') \]

**FAss**

**lemma** \textbf{FAssReds1}:  
\[ P \vdash \langle e, s \rangle \Rightarrow \langle e', s' \rangle \quad \Rightarrow \quad P \vdash (e \cdot F\{D\} := e_2, s) \Rightarrow (e' \cdot F\{D\} := e_2, s') \]

**lemma** \textbf{FAssReds2}:  
\[ P \vdash \langle e, s \rangle \Rightarrow \langle e', s' \rangle \quad \Rightarrow \quad P \vdash (V \cdot F\{D\} := e, s) \Rightarrow (V \cdot F\{D\} := e', s') \]

**lemma** \textbf{FAssRedsVal}: 
\[ \begin{array}{l}
P \vdash \langle e_1, s_0 \rangle \Rightarrow (\text{addr } a, s_1); P \vdash \langle e_2, s_1 \rangle \Rightarrow (\text{Val } v, (h_2, l_2)); \text{ Some } (C, fs) = h_2 a
\end{array} \]
\[ \Rightarrow P \vdash (e_1 \cdot F\{D\} := e_2, s_0) \Rightarrow (\text{unit }, (h_2(a \rightarrow (C, fs(F, D) \Rightarrow v)), l_2)) \]

**lemma** \textbf{FAssRedsNull}: 
\[ \begin{array}{l}
P \vdash \langle e_1, s_0 \rangle \Rightarrow (\text{null}, s_1); P \vdash \langle e_2, s_1 \rangle \Rightarrow (\text{Val } v, s_2)
\end{array} \]
\[ \Rightarrow P \vdash (e_1 \cdot F\{D\} := e_2, s_0) \Rightarrow (\text{THROW NullPointer}, s_2) \]

**lemma** \textbf{FAssRedsThrow1}: 
\[ P \vdash \langle e, s \rangle \Rightarrow (\text{throw } e', s') \quad \Rightarrow \quad P \vdash (e \cdot F\{D\} := e_2, s) \Rightarrow (\text{throw } e', s') \]

**lemma** \textbf{FAssRedsThrow2}: 

\[ P \vdash \langle e_1, s_0 \rangle \Rightarrow \langle \text{Val } v_1, s_1 \rangle; P \vdash \langle e_2, s_1 \rangle \Rightarrow \langle \text{throw } e, s_2 \rangle \]
\[ \Rightarrow P \vdash \langle e_1 \cdot F(D):=e_2, s_0 \rangle \Rightarrow \langle \text{throw } e, s_2 \rangle \]

;;

**Lemma SeqReds:**
\[ P \vdash \langle e, s \rangle \Rightarrow \langle e', s' \rangle \Rightarrow P \vdash \langle e; e_2, s \rangle \Rightarrow \langle e'; e_2, s' \rangle \]

**Lemma SeqReds2:**
\[ P \vdash \langle e, s_0 \rangle \Rightarrow \langle \text{Val } v_1, s_1 \rangle; P \vdash \langle e_2, s_1 \rangle \Rightarrow \langle e_2', s_2 \rangle \]
\[ \Rightarrow P \vdash \langle e_1; e_2, s_0 \rangle \Rightarrow \langle e_2', s_2 \rangle \]

If

**Lemma CondReds:**
\[ P \vdash \langle e, s \rangle \Rightarrow \langle e', s' \rangle \Rightarrow P \vdash \langle \text{if } (e) e_1 \text{ else } e_2, s \rangle \Rightarrow \langle \text{if } (e') e_1 \text{ else } e_2, s' \rangle \]

**Lemma CondReds2T:**
\[ \llbracket P \vdash \langle e, s_0 \rangle \Rightarrow \langle \text{true}, s_1 \rangle; P \vdash \langle e_1, s_1 \rangle \Rightarrow \langle e_1', s_2 \rangle \rrbracket \Rightarrow P \vdash \langle \text{if } (e) e_1 \text{ else } e_2, s_0 \rangle \Rightarrow \langle e_2', s_2 \rangle \]

**Lemma CondReds2F:**
\[ \llbracket P \vdash \langle e, s_0 \rangle \Rightarrow \langle \text{false}, s_1 \rangle; P \vdash \langle e_2, s_1 \rangle \Rightarrow \langle e_2', s_2 \rangle \rrbracket \Rightarrow P \vdash \langle \text{if } (e) e_1 \text{ else } e_2, s_0 \rangle \Rightarrow \langle e_2', s_2 \rangle \]

While

**Lemma WhileFReds:**
\[ P \vdash \langle b, s \rangle \Rightarrow \langle \text{false}, s' \rangle \Rightarrow P \vdash \langle \text{while } (b) \; c, s \rangle \Rightarrow \langle \text{unit}, s' \rangle \]

**Lemma WhileTReds:**
\[ \llbracket P \vdash \langle b, s_0 \rangle \Rightarrow \langle \text{true}, s_1 \rangle; P \vdash \langle e_1, s_1 \rangle \Rightarrow \langle \text{Val } v_1, s_2 \rangle; P \vdash \langle \text{while } (b) \; c, s_2 \rangle \Rightarrow \langle e, s_3 \rangle \rrbracket \]
\[ \Rightarrow P \vdash \langle \text{while } (b) \; c, s_0 \rangle \Rightarrow \langle e, s_3 \rangle \]

**Lemma WhileTReds2:**
\[ \llbracket P \vdash \langle b, s_0 \rangle \Rightarrow \langle \text{true}, s_1 \rangle; P \vdash \langle e_1, s_1 \rangle \Rightarrow \langle \text{throw } e, s_2 \rangle \rrbracket \]
\[ \Rightarrow P \vdash \langle \text{while } (b) \; c, s_0 \rangle \Rightarrow \langle \text{throw } e, s_2 \rangle \]

Throw

**Lemma ThrowReds:**
\[ P \vdash \langle e, s \rangle \Rightarrow \langle e', s' \rangle \Rightarrow P \vdash \langle \text{throw } e, s \rangle \Rightarrow \langle \text{throw } e', s' \rangle \]

**Lemma ThrowRedsNull:**
\[ P \vdash \langle e, s \rangle \Rightarrow \langle \text{null}, s' \rangle \Rightarrow P \vdash \langle \text{throw } e, s \rangle \Rightarrow \langle \text{THROW } \text{NullPointerException}, s' \rangle \]

**Lemma ThrowReds2:**
\[ P \vdash \langle e, s \rangle \Rightarrow \langle \text{throw } a, s' \rangle \Rightarrow P \vdash \langle \text{throw } e, s \rangle \Rightarrow \langle \text{throw } a, s' \rangle \]

InitBlock

**Lemma InitBlockReds-aux:**
\[ P \vdash \langle e, s \rangle \Rightarrow \langle e', s' \rangle \Rightarrow \forall h l h' l' \; v \; s = (h,l(V\rightarrow v)) \Rightarrow s' = (h',l') \Rightarrow P \vdash \langle \langle V; T := \text{Val } v; e \rangle, (h,l) \rangle \Rightarrow \langle \langle V; T := \text{Val } (\text{the}(l' V)); e' \rangle, (h',l'(V:=l V)) \rangle \]

**Lemma InitBlockReds:**
\[ P \vdash \langle e, (h,l(V\rightarrow v)) \rangle \Rightarrow \langle e', (h',l') \rangle \Rightarrow P \vdash \langle \langle V; T := \text{Val } v; e \rangle, (h,l) \rangle \Rightarrow \langle \langle V; T := \text{Val } (\text{the}(l' V)); e' \rangle, (h',l'(V:=l V)) \rangle \]

**Lemma InitBlockRedsFinal:**
Theory Equivalence

\[
\begin{align*}
\[ P \vdash \langle e, (h, l(V \mapsto v)) \rangle \rightarrow \langle e', (h', l') \rangle; \text{ final } e' \] \implies 
\end{align*}
\]

Block

lemma BlockRedsFinal:

assumes \( \text{reds: } P \vdash (e_0, s_0) \rightarrow (e_2, (h_2, l_2)) \) and \( \text{fin: } \text{final } e_2 \)
shows \( \forall h_0, l_0. s_0 = (h_0, l_0(V := \text{None})) \implies P \vdash \langle\{ V; T := \text{Val}\ v; e\},(h, l) \rangle \rightarrow \langle e', (h', l(V := l V)) \rangle \)

try-catch

lemma TryReds:

\( P \vdash \langle e, s \rangle \rightarrow \langle e', s' \rangle \implies P \vdash (\text{try } e \text{ catch } (C \ V) \ e_2, s) \rightarrow (\text{try } e' \text{ catch } (C \ V) \ e_2, s) \)

lemma TryRedsVal:

\( P \vdash \langle e, s \rangle \rightarrow \langle (\text{Val } v, s) \implies P \vdash (\text{try } e \text{ catch } (C \ V) \ e_2, s) \rightarrow (\text{Val } v, s) \)

lemma TryCatchRedsFinal:

\[
\begin{align*}
\[ P \vdash \langle e_1, s_0 \rangle \rightarrow \langle (\text{Throw } a, (h_1, l_1)) \rangle; \ h_1 a = \text{Some}(D, s); P \vdash D \preceq C; \ P \vdash \langle e_2, (h_2, l_2) \rangle \rightarrow \langle e_2', (h_2, l_2(V := l_1 V)) \rangle \implies P \vdash (\text{try } e_1 \text{ catch } (C \ V) \ e_2, s_0) \rightarrow (\text{try } e_2', (h_2, l_2(V := l_1 V))) \]
\end{align*}
\]

lemma TryRedsFail:

\[
\begin{align*}
\[ P \vdash \langle e_1, s \rangle \rightarrow \langle (\text{Throw } a, (h, l)) \rangle; \ h a = \text{Some}(D, s); \neg P \vdash D \preceq C \] \implies P \vdash (\text{try } e_1 \text{ catch } (C \ V) \ e_2, s) \rightarrow (\text{Throw } a, (h, l)) \]
\end{align*}
\]

List

lemma ListReds1:

\( P \vdash \langle e, s \rangle \rightarrow \langle e', s' \rangle \implies P \vdash (e \# e, s, s) \rightarrow (e' \# e, s, s') \)

lemma ListReds2:

\( P \vdash \langle e, s \rangle \rightarrow (e', s, s) \implies P \vdash (\text{Val } v \# e, s) \rightarrow (\text{Val } v \# e', s') \)

lemma ListRedsVal:

\[
\begin{align*}
\[ P \vdash \langle e, s_0 \rangle \rightarrow \langle (\text{Val } v, s_1) \rangle; P \vdash \langle e, s_1 \rangle \rightarrow (e', s_2) \] \implies P \vdash (\text{Val } v \# e, s_0) \rightarrow (\text{Val } v \# e', s_2') \]
\end{align*}
\]

Call

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

lemma assumes \( \text{wf: } \text{wwf-J-prog } P \)
shows \( \text{Red-fv: } P \vdash \langle e, (h, l) \rangle \rightarrow \langle e', (h', l') \rangle \implies \text{fv } e' \subseteq \text{fv } e \)
and \( P \vdash \langle e, (h, l) \rangle \rightarrow (e', (h', l')) \implies \text{fus } e' \subseteq \text{fus } e \)

lemma Red-dom-lcl:

\( P \vdash \langle e, (h, l) \rangle \rightarrow (e', (h', l')) \implies \text{dom } l' \subseteq \text{dom } l \cup \text{fv } e \) and \( P \vdash \langle e, (h, l) \rangle \rightarrow (e', (h', l')) \implies \text{dom } l' \subseteq \text{dom } l \cup \text{fus } e \)

lemma Red-dom-lcl-

\[
\begin{align*}
\[ \text{wwf-J-prog } P; P \vdash (e, (h, l)) \rightarrow (e', (h', l')) \] \implies \text{dom } l' \subseteq \text{dom } l \cup \text{fv } e \]
\end{align*}
\]

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

lemma override-on-upd-lemma:

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

lemma blocksReds:

\( \forall l. \ [\text{length } Vs = \text{length } Ts; \text{length } vs = \text{length } Ts; \text{distinct } Vs; \ P \vdash \langle e, (h, l(Vs \mapsto vs)) \rangle \rightarrow (e', (h', l'))] \)
This direction was carried out by Norbert Schirmer and Daniel Wasserrab.

The big step equivalent of RedWhile:

lemma unfold-while:

\[ \text{lemma} \ \text{blocksFinal:} \]
\[ \forall l. \ [\text{length} \ Vs = \text{length} \ Ts; \text{length} \ vs = \text{length} \ Ts; \text{final} \ e] \implies \]
\[ P \vdash \langle \text{blocks}(Vs,Ts,vs,e), (h,l) \rangle \rightarrow* \langle \text{blocks}(Vs,Ts,\text{map} \ (\text{the} \circ \ l') \ Vs,e'), (h',\text{override-on} \ l' \ l \ (\text{set} \ Vs)) \rangle \]

lemma blocksFinal:

\[ P \vdash \langle \text{blocks}(Vs,Ts,vs,e), (h,l) \rangle \rightarrow* \langle e, (h,l) \rangle \]

lemma blocksRedsFinal:

assumes \( \text{wff}: \text{length} \ Vs = \text{length} \ Ts; \text{length} \ vs = \text{length} \ Ts \) distinct \( Vs \)

and \( \text{reds}: P \vdash \langle e, (h,l(Vs \rightarrow vs)) \rangle \rightarrow* \langle e', (h',l') \rangle \)

and \( \text{fin}: \text{final} \ e' \ \text{and} \ l'' = \text{override-on} \ l' \ l \ (\text{set} \ Vs) \)

shows \( P \vdash \langle \text{blocks}(Vs,Ts,vs,e), (h,l) \rangle \rightarrow* \langle e', (h',l'') \rangle \)

An now the actual method call reduction lemmas.

lemma CallRedsObj:

\[ P \vdash \langle e,s \rangle \rightarrow* \langle e',s' \rangle \implies P \vdash \langle e \cdot M(es), s \rangle \rightarrow* \langle e' \cdot M(es), s' \rangle \]

lemma CallRedsParams:

\[ P \vdash \langle es,s \rangle \rightarrow* \langle es',s' \rangle \implies P \vdash \langle (\text{Val} \ v) \cdot M(es), s \rangle \rightarrow* \langle (\text{Val} \ v) \cdot M(es'), s' \rangle \]

lemma CallRedsFinal:

assumes \( \text{wff}: \text{wwf-J-prog} \ P \)

and \[ P \vdash \langle e,s_0 \rangle \rightarrow* \langle \text{addr} \ a,s_1 \rangle \]

\[ P \vdash \langle es,s_1 \rangle \rightarrow* \langle \text{map} \ Val \ vs,(h_2,l_2) \rangle \]

\[ h_2 \ a = \text{Some}(C,fs) \ \text{P} \vdash \text{C sees} \ M : Ts \rightarrow T = (\text{pns}, \text{body}) \] \text{in} \ D \]

size \( vs = \text{size} \ pns \)

and \( l_2' : l_2' = [\text{this} \rightarrow \text{Addr} \ a, \text{pns}[\rightarrow]vs] \)

and \( \text{body}: P \vdash \langle \text{body},(h_2,l_2') \rangle \rightarrow* \langle ef,(h_3,l_3) \rangle \)

and \( \text{final} \ ef \)

shows \( P \vdash \langle e \cdot M(es), s_0 \rangle \rightarrow* \langle ef,(h_3,l_2) \rangle \)

lemma CallRedsThrowParams:

\[ \langle P \vdash \langle e,s \rangle \rightarrow* \langle \text{Val} \ v,s_1 \rangle; \ P \vdash \langle es,s_1 \rangle \rightarrow* \langle \text{map} \ Val \ vs_1 \ @ \ \text{throw} \ a \ # \ es_2,s_2 \rangle \ \rangle \implies P \vdash \langle e \cdot M(es), s_0 \rangle \rightarrow* \langle \text{throw} \ a,s_2 \rangle \]

lemma CallRedsThrowObj:

\[ P \vdash \langle e,s_0 \rangle \rightarrow* \langle \text{throw} \ a,s_1 \rangle \implies P \vdash \langle e \cdot M(es), s_0 \rangle \rightarrow* \langle \text{throw} \ a,s_1 \rangle \]

lemma CallRedsNull:

\[ \langle P \vdash \langle e,s_0 \rangle \rightarrow* \langle \text{null},s_1 \rangle; \ P \vdash \langle es,s_1 \rangle \rightarrow* \langle \text{map} \ Val \ vs,s_2 \rangle \ \rangle \implies P \vdash \langle e \cdot M(es), s_0 \rangle \rightarrow* \langle \text{THROW} \ \text{NullPointer},s_2 \rangle \]

The Main Theorem

lemma assumes \( \text{wff}: \text{wwf-J-prog} \ P \)

shows \( \text{big-by-small}: P \vdash \langle e,s \rangle \Rightarrow \langle e',s' \rangle \implies P \vdash \langle e,s \rangle \rightarrow* \langle e',s' \rangle \)

and \( \text{biggs-by-smalls}: P \vdash \langle es,s \rangle \Rightarrow \langle es',s' \rangle \implies P \vdash \langle es,s \rangle \rightarrow* \langle es',s' \rangle \)

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:
Theory Equivalence

\[ P \vdash (\text{while}(b) \ c, s) \Rightarrow \langle e', s' \rangle = P \vdash (\text{if}(b) (e; \text{while}(b) \ c) \ \text{else} \ (\text{unit}), s) \Rightarrow \langle e', s' \rangle \]

**Lemma** blocksEval:
\[ \forall T_s \ vs \ l \ l'. \ [\text{size } ps = \text{size } Ts; \ \text{size } vs = \text{size } vs] \ P \vdash \langle \text{blocks}(ps, Ts, vs, e), (h, l) \rangle \Rightarrow \langle e', (h', l') \rangle \]
\[ \Rightarrow \exists \ l''. \ P \vdash \langle e, (h, l[\Rightarrow \langle vs \rangle]) \Rightarrow \langle e', (h', l'') \rangle \]

**Lemma** assumes wf: wuf-J-prog P
**SHOWS** eval-restrict-lcl:
\[ P \vdash \langle e, (h, l) \rangle \Rightarrow \langle e', (h', l') \rangle \Rightarrow (\forall W. \ \text{fv } e \subseteq W \Rightarrow P \vdash \langle e, (h, l \cdot W) \rangle \Rightarrow \langle e', (h', l') \cdot W) \rangle) \]
\[ \text{and } P \vdash \langle es, (h, l) \rangle \Rightarrow \langle es', (h', l') \rangle \Rightarrow (\forall W. \ \text{fvs } es \subseteq W \Rightarrow P \vdash \langle es, (h, l \cdot W) \rangle \Rightarrow \langle es', (h', l') \cdot W) \rangle) \]

**Lemma** eval-notfree-unchanged:
\[ P \vdash \langle e, (h, l) \rangle \Rightarrow \langle e', (h', l') \rangle \Rightarrow (\forall V. \ V \not\in \text{fv } e \Rightarrow l' \ V = l \ V) \]
\[ \text{and } P \vdash \langle es, (h, l) \rangle \Rightarrow \langle es', (h', l') \rangle \Rightarrow (\forall V. \ V \not\in \text{fes } es \Rightarrow l' \ V = l \ V) \]

**Lemma** eval-closed-lcl-unchanged:
\[ \forall P \vdash \langle e, (h, l) \rangle \Rightarrow \langle e', (h', l') \rangle; \ \text{fv } e = \{ \} \Rightarrow l' = l \]

**Lemma** list-eval-Throw:
**ASSUMES** eval-e: P \vdash \langle \text{throw } x, s \rangle \Rightarrow \langle e', s' \rangle
**SHOWS** P \vdash \langle \text{map } \text{Val } vs @ \text{throw } x \ # \ es', s' \rangle \Rightarrow \langle \text{map } \text{Val } vs @ e' \ # \ es', s' \rangle

The key lemma:

**Lemma** assumes wf: wuf-J-prog P
**SHOWS** extend-1-eval:
\[ P \vdash \langle e, s \rangle \Rightarrow \langle e', s' \rangle \Rightarrow (\forall t'. \ e \Rightarrow t' \ \Rightarrow P \vdash \langle e, s \rangle \Rightarrow \langle e', s' \rangle) \]
\[ \text{and } extend-1-evals:
\[ P \vdash \langle e, t \rangle \Rightarrow \langle e', t' \rangle \Rightarrow (\forall t'. \ (e \Rightarrow t') \Rightarrow P \vdash \langle e, t \rangle \Rightarrow \langle e', t' \rangle) \]

Its extension to \( \to^* \):

**Lemma** extend-eval:
**ASSUMES** wf: wuf-J-prog P
**AND** reds: P \vdash \langle e, s \rangle \Rightarrow^* \langle e'', s'' \rangle \text{ 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: wuf-J-prog P
**AND** reds: P \vdash \langle es, s \rangle \Rightarrow^* \langle es'', s'' \rangle \text{ 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: wuf-J-prog P
**SHOWS** small-by-big: [P \vdash \langle e, s \rangle \Rightarrow^* \langle e', s' \rangle; \ \text{final } es \Rightarrow P \vdash \langle e, s \rangle \Rightarrow \langle e', s' \rangle]
\[ \text{and } P \vdash \langle es, s \rangle \Rightarrow^* \langle es', s' \rangle; \ \text{finals } es \Rightarrow P \vdash \langle es, s \rangle \Rightarrow \langle es', s' \rangle] \]

2.15.3 Equivalence

And now, the crowning achievement:

**Corollary** big-iff-small:
wuf-J-prog P \Rightarrow
\[ P \vdash \langle e, s \rangle \Rightarrow \langle e', s' \rangle = (P \vdash \langle e, s \rangle \Rightarrow^* \langle e', s' \rangle \land \text{final } e') \]
2.16 Well-typedness of Jinja expressions

theory WellType
imports ../Common/Objects Expr
begin

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 \lor 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 ⊢ e1 :: T1; P,E ⊢ e2 :: T2; P ⊢ T1 ≤ T2 \lor P ⊢ T2 ≤ T1 ] ]
⇒ P,E ⊢ e1 ≪Eq≫ e2 :: Boolean

| WTBinOpAdd:
[ [ P,E ⊢ e1 :: Integer; P,E ⊢ e2 :: Integer ] ]
⇒ P,E ⊢ e1 ≪Add≫ e2 :: Integer

| WTLAss:
[ [ E V = Some T; P,E ⊢ e :: T'; P ⊢ T ≤ T; V ≠ this ] ]
⇒ P,E ⊢ V:=e :: Void

| WFTAcc:
[ [ P,E ⊢ e :: Class C; P ⊢ C sees F:T in D ] ]
⇒ P,E ⊢ e·F{D} :: T

| WFTAss:
\[
\begin{array}{l}
\text{Lemma: well-typed expression lists}\\
\end{array}
\]
\[ P, E \vdash es :: Ts \implies (\forall E'. E \subseteq_m E' \implies P, E' \vdash es :: Ts) \]

**lemma WT-fv:** \(P, E \vdash e :: T \implies \text{fv} e \subseteq \text{dom} E\)

and \(P, E \vdash es :: Ts \implies \text{fvs} es \subseteq \text{dom} E\)

### 2.17 Runtime Well-typedness

**theory** WellTypeRT

**imports** WellType

**begin**

**inductive**

\(\text{WTrt} :: \text{J-prog} \Rightarrow \text{heap} \Rightarrow \text{env} \Rightarrow \text{expr} \Rightarrow \text{ty} \Rightarrow \text{bool} \)

and \(\text{WTrts} :: \text{J-prog} \Rightarrow \text{heap} \Rightarrow \text{env} \Rightarrow \text{expr list} \Rightarrow \text{ty list} \Rightarrow \text{bool} \)

and \(\text{WTrt2} :: [\text{J-prog}, \text{env}, \text{heap}, \text{expr}, \text{ty}] \Rightarrow \text{bool} \)

(-, -, -) \(\vdash - : [51, 51, 51]50 \)

and \(\text{WTrts2} :: [\text{J-prog}, \text{env}, \text{heap}, \text{expr list}, \text{ty list}] \Rightarrow \text{bool} \)

(-, -, -) \(\vdash - [:] : [51, 51, 51]50 \)

for \(P :: \text{J-prog}\) and \(h :: \text{heap}\)

**where**

\(P, E, h \vdash e : T \equiv \text{WTrt} P h E e T\)

\(| P, E, h \vdash es :: Ts \equiv \text{WTrts} P h E es Ts\)

| \(\text{WTrtNew}::\)
| is-class \(P C \implies P, E, h \vdash \text{new} C : \text{Class} C\)

| \(\text{WTrtCast}::\)
| \[ \text{typeof}_{h} v = \text{Some} T \implies P, E, h \vdash \text{Val} v : T\]

| \(\text{WTrtVar}::\)
| \[ E V = \text{Some} T \implies P, E, h \vdash \text{Var} V : T\]

| \(\text{WTrtBinOpEq}::\)
| \[ P, E, h \vdash e_1 : T_1; P, E, h \vdash e_2 : T_2 \] \implies P, E, h \vdash e_1 \ll Eq \gg e_2 : \text{Boolean}\)

| \(\text{WTrtBinOpAdd}::\)
| \[ P, E, h \vdash e_1 : \text{Integer}; P, E, h \vdash e_2 : \text{Integer} \] \implies P, E, h \vdash e_1 \ll Add \gg e_2 : \text{Integer}\)

| \(\text{WTrtLAss}::\)
| \[ E V = \text{Some} T; P, E, h \vdash e : T'; P \vdash T' \leq T \] \implies P, E, h \vdash V := e : \text{Void}\)

| \(\text{WTrtFAcc}::\)
\[ \[ P, E, h \vdash e : \text{Class C}; P \vdash C \text{ has } F : T \text{ in } D \] \implies P, E, h \vdash e \cdot F(D) : T \]

| WTrfAssNT: |
| \[ P, E, h \vdash e : NT \] |
| \[ P, E, h \vdash e \cdot F(D) : T \] |

| WTrfAss: |
| \[ P, E, h \vdash e_1 : \text{Class C}; P \vdash C \text{ has } F : T \text{ in } D; P, E, h \vdash e_2 : T_2; P \vdash T_2 \leq T \] |
| \[ \implies P, E, h \vdash e_1 \cdot F(D) ;= e_2 : \text{Void} \] |

| WTrfAssNT: |
| \[ P, E, h \vdash e_1 : NT; P, E, h \vdash e_2 : T_2 \] |
| \[ \implies P, E, h \vdash e_1 \cdot F(D) ;= e_2 : \text{Void} \] |

| WTrCall: |
| \[ P, E, h \vdash e : \text{Class C}; P \vdash C \text{ sees } M \cdot Ts \rightarrow T = (\text{pns}, \text{body}) \text{ in } D; \] |
| \[ P, E, h \vdash e \cdot M(\text{es}) : T \] |

| WTrCallNT: |
| \[ P, E, h \vdash e \cdot M(\text{es}) : T \] |

| WTrBlock: |
| \[ P, E(V \mapsto T), h \vdash e : T' \] |
| \[ \implies P, E, h \vdash \{ V : T; e \} : T' \] |

| WTrCond: |
| \[ P, E, h \vdash e : \text{Boolean}; P, E, h \vdash e_1 : T_1; P, E, h \vdash e_2 : T_2; \] |
| \[ P \vdash T_1 \leq T_2 \lor P \vdash T_2 \leq T_1; P \vdash T_1 \leq T_2 \implies T = T_2; P \vdash T_2 \leq T_1 \implies T = T_1 \] |
| \[ \implies P, E, h \vdash \text{if } (e) e_1 \text{ else } e_2 : T \] |

| WTrWhile: |
| \[ P, E, h \vdash e : \text{Boolean}; P, E, h \vdash c : T \] |
| \[ \implies P, E, h \vdash \text{while}(e) c : T \] |

| WTrThrow: |
| \[ P, E, h \vdash e : T_e; \text{is-refT } T_e \] |
| \[ \implies P, E, h \vdash \text{throw } e : T \] |

| WTrTry: |
| \[ P, E, h \vdash e_1 : T_1; P, E(V \mapsto \text{Class C}), h \vdash e_2 : T_2; P \vdash T_1 \leq T_2 \] |
| \[ \implies P, E, h \vdash \text{try } e_1 \text{ catch}(C V) e_2 : T_2 \] |

— well-typed expression lists

| WTrNil: |
| \[ P, E, h \vdash \text{[]} \] |
2.18.1 Hypersets

definition hyperUn :: 'a hyperset ⇒ 'a hyperset ⇒ 'a hyperset  (infixl ⊔ 65) where
  A ⊔ B ≡ case A of None ⇒ None
  | [A] ⇒ (case B of None ⇒ None | [B] ⇒ [A ⊔ B])

definition hyperInt :: 'a hyperset ⇒ 'a hyperset ⇒ 'a hyperset  (infixl ∩ 70) where
  A ∩ B ≡ case A of None ⇒ B
  | [A] ⇒ (case B of None ⇒ [A] | [B] ⇒ [A ∩ B])

2.17.1 Easy consequences

lemma [iff]: (P,E,h ⊢ [] [:] Ts) = (Ts = [])

lemma [iff]: (P,E,h ⊢ e # es [:] T # Ts) = (P,E,h ⊢ e : T ∧ P,E,h ⊢ es [:] Ts)

lemma [iff]: (P,E,h ⊢ (e # es) [:] Ts)
  = (∃ U Us. Ts = U # Us ∧ P,E,h ⊢ e : U ∧ P,E,h ⊢ es [:] Us)

lemma [simp]: ∀ Ts. (P,E,h ⊢ es₁ ⊔ es₂ [:] Ts) =
  (∃ Ts₁ Ts₂. Ts = Ts₁ ⊔ Ts₂ ∧ P,E,h ⊢ es₁ [:] Ts₁ & P,E,h ⊢ es₂ [:] Ts₂)

lemma [iff]: P,E,h ⊢ Val v : T = (typeof_h v = Some T)

lemma [iff]: P,E,h ⊢ Var v : T = (E v = Some T)

lemma [iff]: P,E,h ⊢ e₁;e₂ : T = (∃ T₁. P,E,h ⊢ e₁ːT₁ ∧ P,E,h ⊢ e₂ːT₂)

lemma [iff]: P,E,h ⊢ {VːT; e} : T' = (P,E(V⇒T),h ⊢ e : T')

2.17.2 Some interesting lemmas

lemma WTrts-Val[simp]:
    ∀ Ts. (P,E,h ⊢ map Val vs [:] Ts) = (map (typeof_h) vs = map Some Ts)

lemma WTrts-same-length: ∀ Ts. P,E,h ⊢ es [:] Ts → length es = length Ts

lemma WTrt-env-mono:
    P,E,h ⊢ e : T =⇒ (∀ E'. E ⊆ₚ E' =⇒ P,E',h ⊢ e : T) and
    P,E,h ⊢ es [:] Ts =⇒ (∀ E'. E ⊆ₚ E' =⇒ P,E',h ⊢ es [:] Ts)

lemma WTrt-hext-mono: P,E,h ⊢ e : T =⇒ h ⊆ h' =⇒ P,E,h' ⊢ e : T
and WTrts-hext-mono: P,E,h ⊢ es [:] Ts =⇒ h ⊆ h' =⇒ P,E,h' ⊢ es [:] Ts

lemma WT-implies-WTrt: P,E ⊢ e :: T =⇒ P,E,h ⊢ e : T
and WT₁-implies-WTrts: P,E ⊢ es :: Ts =⇒ P,E,h ⊢ es :: Ts

end

2.18 Definite assignment

theory DefAss imports BigStep begin

2.18.1 Hypersets

type-synonym 'a hyperset = 'a set option

definition hyperUn :: 'a hyperset ⇒ 'a hyperset ⇒ 'a hyperset  (infixl ⊔ 65) where
  A ⊔ B ≡ case A of None ⇒ None
  | [A] ⇒ (case B of None ⇒ None | [B] ⇒ [A ⊔ B])

definition hyperInt :: 'a hyperset ⇒ 'a hyperset ⇒ 'a hyperset  (infixl ∩ 70) where
  A ∩ B ≡ case A of None ⇒ B
  | [A] ⇒ (case B of None ⇒ [A] | [B] ⇒ [A ∩ B])
definition hyperDiff1 :: 'a hyperset ⇒ 'a ⇒ 'a hyperset (infixl ⊖ 65)
where
  A ⊖ a ≡ case A of None ⇒ None | [A] ⇒ [A − {a}]

definition hyper-isin :: 'a ⇒ 'a hyperset ⇒ bool (infix ∈∈ 50)
where
  a ∈∈ A ≡ case A of None ⇒ True | [A] ⇒ a ∈ A

definition hyper-subset :: 'a hyperset ⇒ 'a hyperset ⇒ bool (infix ⊑ 50)
where
  A ⊑ B ≡ case B of None ⇒ True | [B] ⇒ (case A of None ⇒ False | [A] ⇒ A ⊆ B)

lemmas hypersetdefs =
hyperUn-def hyperInt-def hyperDiff1-def hyper-isin-def hyper-subset-def

lemma [simp]: [{a}] ⊔ A = A ∧ A ⊔ [{a}] = A
lemma [simp]: [A] ⊔ [B] = [A ∪ B] ∧ [A] ⊖ a = [A − {a}]
lemma [simp]: None ⊔ A = None ∧ A ⊔ None = None
lemma [simp]: a ∈∈ None ∧ None ⊖ a = None
lemma hyperUn-assoc: (A ⊔ B) ⊔ C = A ⊔ (B ⊔ C)
lemma hyper-insert-comm: A ⊔ [{a}] = [{a}] ⊔ A ∧ A ⊔ ([{a}] ⊔ B) = [{a}] ⊔ (A ⊔ B)

2.18.2 Definite assignment

primrec A :: 'a exp ⇒ 'a hyperset
  and As :: 'a exp list ⇒ 'a hyperset
where
  A (new C) = [{}]
| A (Cast C e) = A e
| A (Val v) = [{}]
| A (e1 <bop> e2) = A e1 ⊔ A e2
| A (Var V) = [{}]
| A (LAss V e) = [{V}] ⊔ A e
| A (e·F{D}) = A e
| A (e·M(\_)) = A e ⊔ As es
| A (\{V:T; e\}) = A e ⊖ V
| A (e1;e2) = A e1 ⊔ A e2
| A (if (e) e1 else e2) = A e ⊔ (A e1 ∩ A e2)
| A (while (b) e) = A b
| A (throw e) = None
| A (try e1 catch(C V) e2) = A e1 ⊖ (A e2 ⊔ V)
| As ([]) = [{}]
| As (e#es) = A e ⊔ As es

primrec D :: 'a exp ⇒ 'a hyperset ⇒ bool
  and Ds :: 'a exp list ⇒ 'a hyperset ⇒ bool
where
  D (new C) A = True
2.19 Conformance Relations for Type Soundness Proofs

theory Conform
imports Exceptions
begin

definition conf :: 'm prog ⇒ heap ⇒ val ⇒ ty ⇒ bool (⊥,⊥ - : ⩽ - [51,51,51,51] 50) where
  P,h ⊢ v ⩽ T ≡
  ∃ T'. typeof h v = Some T' ∧ P ⊢ T' ⩽ T

definition oconf :: 'm prog ⇒ heap ⇒ obj ⇒ bool (⊥,⊥ - √ [51,51,51] 50) where
  P,h ⊢ obj √ ≡
  let (C,fs) = obj in ∀ F D T. P ⊢ C has F:T in D →
  (∃ v. fs(F,D) = Some v ∧ P,h ⊢ v ⩽ T)
Theory Conform

definition hconf :: 'm prog ⇒ heap ⇒ bool  (\vdash - (√ [51,51]) ≤)
where
P \vdash h (√) ≡ 
(\forall a \text{ obj. } h a = \text{ Some obj } \rightarrow P,h \vdash (√) ) \land \text{ preallocated } h

definition lconf :: 'm prog ⇒ heap ⇒ (\text{vname} \rightarrow \text{val}) ⇒ (\text{vname} \rightarrow \text{ty}) ⇒ bool  (\vdash - (\vdash ':) - (√ [51,51,51,51]) ≤)
where
P,h \vdash l (≤) E ≡ 
\forall V v. l V = \text{ Some v } \rightarrow (\exists T. E V = \text{ Some T } \land P,h \vdash l (≤) T)

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

2.19.1 Value conformance :≤

lemma conf-Null [simp]: P,h ⊢ Null (≤) T = P \vdash NT (≤) T
lemma typedef-conf [simp]: typedef v = \text{ Some T } \rightarrow P,h \vdash v (≤) T
lemma typedef-lit-conf [simp]: typedef v = \text{ Some T } \rightarrow P,h \vdash v (≤) T
lemma defeal-conf [simp]: P,h \vdash default-val T (≤) T
lemma conf-upd-obj: h a = Some(C,fs) \rightarrow (P,h(a→(C,fs))) (≤) x = (P,h (≤) T)
lemma conf-widen: P,h \vdash v (≤) T \rightarrow P \vdash T (≤) T' \rightarrow P,h \vdash v (≤) T'
lemma conf-hext: h (≤) h' \rightarrow P,h \vdash v (≤) T \rightarrow P,h' (≤) v (≤) T
lemma conf-ClassD: P,h \vdash v (≤) Class C \rightarrow 
\forall a \text{ obj. } h a = \text{ Some } (C,fs) \land P,h \vdash (C \land v (≤) Class C)
lemma conf-NT [iff]: P,h \vdash v (≤) NT = (v = Null)
lemma non-npD: [ v \neq Null; P,h \vdash v (≤) Class C ] \rightarrow \exists a C' fs. v = Addr a \land h a = Some(C',fs) \land P (\vdash - (C' \vdash ':) - (√" C

2.19.2 Value list conformance [:≤]

lemma confs-widens [trans]: [P,h \vdash vs (≤) Ts; P \vdash Ts (≤) Ts'] \rightarrow P,h \vdash vs (≤) Ts'
lemma confs-rev: P,h \vdash rev s (≤) t = (P,h \vdash s (≤) rev t)
lemma confs-conv-map:
\bigwedge Ts'. P,h \vdash vs (≤) Ts' = (\exists Ts. map typedef v = map Some Ts \land P \vdash Ts (≤) Ts')
lemma confs-hext: P,h \vdash vs (≤) Ts \rightarrow h (≤) h' \rightarrow P,h' \vdash vs (≤) Ts
lemma confs-Cons2: P,h \vdash vs (≤) ys = (\exists zs xs z y. z (≤) y \land P,h \vdash zs (≤) y)

2.19.3 Object conformance

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

lemma oconf-fupd [intro?]:
\[ P \vdash C \text{ has } F:T \text{ in } D; P,h \vdash v (≤) T; P,h \vdash (C,fs) (\vdash) \]
\[ \rightarrow P,h \vdash (C, fs((F,D)\rightarrow v)) (\vdash) \]

2.19.4 Heap conformance

lemma hconfD: [ P \vdash h (\vdash); h a = Some obj ] \rightarrow P,h \vdash obj (\vdash)
lemma \texttt{hconf-new}: \[ P \vdash h \land h \text{ is } \texttt{None}; P, h \vdash \text{ obj } \land \] \[ \Rightarrow P \vdash h(a \rightarrow \text{ obj}) \land \]

lemma \texttt{hconf-upd}: \[ P, h \vdash l (\leq) E; h \leq h' \] \[ \Rightarrow P, h' \vdash l (\leq) E \]

2.19.5 Local variable conformance

lemma \texttt{lconf-hext}: \[ P, h \vdash l (\leq) E; P, h \vdash v \leq T; E V = \texttt{Some} T \] \[ \Rightarrow P, h \vdash l(V \rightarrow v) (\leq) E \]

lemma \texttt{lconf-upd}: \[ P, h \vdash l (\leq) E; P, h \vdash v :< T \] \[ \Rightarrow P, h \vdash l(V \rightarrow v) (\leq) E(V \rightarrow T) \]

end

2.20 Progress of Small Step Semantics

theory \texttt{Progress}

imports \texttt{Equivalence WellTypeRT DefAss ..} ./\texttt{Common/Conform}

begin

lemma \texttt{final-addrE}:
\[
\begin{align*}
\forall a. e = \text{ addr } a & \Rightarrow R; \\
\forall a. e = \text{ Throw } a & \Rightarrow R \implies R
\end{align*}
\]

lemma \texttt{finalRefE}:
\[
\begin{align*}
\forall a. C. \[ e = \text{ addr } a; T = \text{ Class } C \] & \Rightarrow R; \\
\forall a. e = \text{ Throw } a & \Rightarrow R \implies R
\end{align*}
\]

Derivation of new induction scheme for well typing:

inductive
\[
\begin{align*}
\text{WTrt'} :: [\text{ J-prog, heap, env, expr, ty }] & \Rightarrow \text{ bool} \\
\text{and WTrts'} :: [\text{ J-prog, heap, env, expr list, ty list }] & \Rightarrow \text{ bool} \\
\text{and WTrt'2} :: [\text{ J-prog, heap, expr, ty }] & \Rightarrow \text{ bool} \\
\text{and WTrts'2} :: [\text{ J-prog, heap, expr list, ty list }] & \Rightarrow \text{ bool}
\end{align*}
\]

for \texttt{P} :: \texttt{J-prog} and \texttt{h} :: \texttt{heap}

where
\[
\begin{align*}
P, E, h \vdash e : T & \equiv \text{ WTrt'} P h e T; \\
P, E, h \vdash e :\text{ new C :'} & \equiv \text{ WTrts'} P h e T
\end{align*}
\]

| is-class P C \Rightarrow P, E, h \vdash \text{ new C :'} Class C \\
| [ P, E, h \vdash e : T; is-ref T T; is-class P C ] \Rightarrow P, E, h \vdash \text{ Cast C :'} Class C \\
| typeof h v = \text{ Some } T \Rightarrow P, E, h \vdash \text{ Val } v :' T \\
| E v = \text{ Some } T \Rightarrow P, E, h \vdash \text{ Var } v :' T \\
| [ P, E, h \vdash e_1 :' T_1; P, E, h \vdash e_2 :' T_2 ] \Rightarrow P, E, h \vdash e_1 <\text{ Eq}> e_2 :' \text{ Boolean} \\
| [ P, E, h \vdash e_1 :' \text{ Integer}; P, E, h \vdash e_2 :' \text{ Integer} ] \Rightarrow P, E, h \vdash e_1 <\text{ Add}> e_2 :' \text{ Integer} \\
| [ P, E, h \vdash \text{ Var } V :' T; P, E, h \vdash e :' T'; P \vdash T' \leq T (\ast V \neq \text{ This*}) ] \Rightarrow P, E, h \vdash V := e :' \text{ Void}
imports ../
begin

Theory JWellForm

lemma [iff]: P,E,h \vdash e : T' \iff P,E,h \vdash e : T

end
**definition** \( \text{wf-J-mdecl} :: J\text{-prog} \Rightarrow \text{cname} \Rightarrow J\text{-mb mdecl} \Rightarrow \text{bool} \)

**where**

\( \text{wf-J-mdecl} \ P \ C \equiv \lambda (M, Ts, T, (\text{pns, body})). \)

\( \text{length} \ Ts = \text{length} \ \text{pns} \land \)

\( \text{distinct} \ \text{pns} \land \)

\( \text{this} \notin \text{set} \ \text{pns} \land \)

\( (\exists \ T'. P, [\text{this} \rightarrow \text{Class} \ C, \text{pns}[\rightarrow] \text{Ts}] \vdash \text{body} :: T' \land P \vdash T' \leq T) \land \)

\( D \ \text{body} \ [(\text{this}) \cup \text{set} \ \text{pns}] \)

**lemma** \( \text{wf-J-mdecl}[\text{simp}]: \)

\( \text{wf-J-mdecl} \ P \ C (M, Ts, T, \text{pns}, \text{body}) \equiv \)

\( (\text{length} \ Ts = \text{length} \ \text{pns} \land \)

\( \text{distinct} \ \text{pns} \land \)

\( (\exists \ T'. P, [\text{this} \rightarrow \text{Class} \ C, \text{pns}[\rightarrow] \text{Ts}] \vdash \text{body} :: T' \land P \vdash T' \leq T) \land \)

\( D \ \text{body} \ [(\text{this}) \cup \text{set} \ \text{pns}] \)

**abbreviation**

\( \text{wf-J-prog} :: J\text{-prog} \Rightarrow \text{bool} \)

**where**

\( \text{wf-J-prog} == \text{wf-prog} \ \text{wf-J-mdecl} \)

**lemma** \( \text{wf-J-prog-wf-J-mdecl}: \)

\[ \[ \text{wf-J-prog} \ P; (C, D, \text{fds}, \text{mths}) \in \text{set} \ P; \text{jmdcl} \in \text{set} \ \text{mths} \] \]

\( \Rightarrow \ \text{wf-J-mdecl} \ P \ C \ \text{jmdcl} \)

**lemma** \( \text{wf-mdecl-wwf-mdecl}: \)

\( \text{wf-J-mdecl} \ P \ C \ \text{Md} \Rightarrow \ \text{wwf-J-mdecl} \ P \ C \ \text{Md} \)

**lemma** \( \text{wf-prog-wwf-prog}: \)

\( \text{wf-J-prog} \ P \Rightarrow \ \text{wwf-J-prog} \ P \)

**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:

\( P \vdash \langle e, (h, l) \rangle \rightarrow \langle e', (h', l') \rangle \Implies (\forall T E. \ [ P, E, h \vdash e : T; P \vdash h \sqrt{\ }] \implies P \vdash h' \sqrt{\ }) \)

**and** reds-preserves-hconf:

\( P \vdash \langle es, (h, l) \rangle \rightarrow\langle es', (h', l') \rangle \implies (\forall Ts E. \ [ P, E, h \vdash es[\vdash] Ts; P \vdash h \sqrt{\ }] \implies P \vdash h' \sqrt{\ }) \)

**theorem** red-preserves-lconf:

\( P \vdash \langle e, (h, l) \rangle \rightarrow \langle e', (h', l') \rangle \implies (\forall T E. \ [ P, E, h \vdash e; T; P \vdash h \sqrt{\leq} E ] \implies P, h \vdash l' \sqrt{\leq} E) \)

**and** reds-preserves-lconf:

\( P \vdash \langle es, (h, l) \rangle \rightarrow \langle es', (h', l') \rangle \implies (\forall Ts E. \ [ P, E, h \vdash es[\vdash] Ts; P, h \vdash l \sqrt{\leq} E ] \implies P, h \vdash l' \sqrt{\leq} E) \)

Preservation of definite assignment more complex and requires a few lemmas first.
lemma [iff]: \( \forall A. \ [ \text{length } Vs = \text{length } Ts; \ \text{length } vs = \text{length } Ts'] \implies \) 
\( D (\text{blocks } (Vs, Ts, vs, e)) A = D e (A \uplus \{\text{set } Vs\}) \)

lemma red-IA-incr: \( P \vdash \langle e, (h, l) \rangle \rightarrow \langle e', (h', l') \rangle \implies [\text{dom } l] \uplus A e \subseteq [\text{dom } l'] \uplus A e' \)
and reds-IA-incr: \( P \vdash \langle es, (h, l) \rangle \rightarrow \langle es', (h', l') \rangle \implies [\text{dom } l] \uplus A \ es \subseteq [\text{dom } l'] \uplus A \ es' \)

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

lemma assumes wf: \( \text{wf-J-prog } P \)
shows red-preserve-defass:
\( P \vdash \langle e, (h, l) \rangle \rightarrow \langle e', (h', l') \rangle \implies D e [\text{dom } l] \rightarrow D e' [\text{dom } l'] \)
and reds-preserve-defass:
\( P \vdash \langle es, (h, l) \rangle \rightarrow \langle es', (h', l') \rangle \implies D s \ es [\text{dom } l] \rightarrow D s' \ es' [\text{dom } l'] \)

Combining conformance of heap and local variables:

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

lemma red-preserve-sconf:
\( [P \vdash \langle e, s \rangle \rightarrow \langle e', s' \rangle; P, E, hp s \vdash e : T; P, E \vdash s \checkmark] \implies P, E \vdash s' \checkmark \)
lemma reds-preserve-sconf:
\( [P \vdash \langle es, s \rangle \rightarrow \langle es', s' \rangle; P, E, hp s \vdash es [:] Ts; P, E \vdash s \checkmark] \implies P, E \vdash s' \checkmark \)

2.22.2 Subject reduction

lemma wt-blocks:
\( \forall E. \ [ \text{length } Vs = \text{length } Ts; \ \text{length } vs = \text{length } Ts'] \implies \) 
\( (P, E, h \vdash \text{blocks } (Vs, Ts, vs, e) : T) = \)
\( (P, E, (Vs [\sim] Ts), h \vdash e : T \land (3 Ts'. \ \text{map } (\text{typeof } h) vs = \text{map } \text{Some } Ts' \land P \vdash Ts' [\leq] Ts)) \)

theorem assumes wf: \( \text{wf-J-prog } P \)
shows subject-reduction2: \( P \vdash \langle e, (h, l) \rangle \rightarrow \langle e', (h', l') \rangle \implies \) 
\( (\forall E. \ P, E \vdash (h, l) \checkmark; P, E, h \vdash e : T) \)
\( \rightarrow \exists T'. \ P, E, h' \vdash e : T' \land P \vdash T' \leq T) \)
and subjects-reduction2:
\( P \vdash \langle es, (h, l) \rangle \rightarrow \langle es', (h', l') \rangle \implies \) 
\( (\forall E. \ P, E \vdash (h, l) \checkmark; P, E, h \vdash es [:] Ts) \)
\( \rightarrow \exists Ts'. \ P, E, h' \vdash es' [:] Ts' \land P \vdash Ts' [\leq] Ts) \)

corollary subject-reduction:
\( \forall \text{wf-J-prog } P; P \vdash \langle e, s \rangle \rightarrow \langle e', s' \rangle; P, E \vdash s \checkmark; P, E, hp s \vdash e : T \) 
\( \rightarrow \exists T'. \ P, E, hp s' \vdash e' : T' \land P \vdash T' \leq T \)

corollary subjects-reduction:
\( \forall \text{wf-J-prog } P; P \vdash \langle es, s \rangle \rightarrow \langle es', s' \rangle; P, E \vdash s \checkmark; P, E, hp s \vdash es [:] Ts \) 
\( \rightarrow \exists Ts'. \ P, E, hp s' \vdash es' [:] Ts' \land P \vdash Ts' [\leq] Ts \)

2.22.3 Lifting to \( \\rightarrow* \)
Now all these preservation lemmas are first lifted to the transitive closure . . .

lemma Red-preserve-sconf:
assumes \( \text{wf-J-prog } P \) \( \text{and } \text{Red} : P \vdash \langle e, s \rangle \rightarrow* \langle e', s' \rangle \)
shows \( \forall T. \ [P, E, hp s \vdash e : T; P, E \vdash s \checkmark] \implies P, E \vdash s' \checkmark \)

lemma Red-preserve-defass:
assumes \( \text{wf-J-prog } P \) \( \text{and } \text{reds} : P \vdash \langle e, s \rangle \rightarrow* \langle e', s' \rangle \)
shows \( D e [\text{dom } (\text{lcl } s)] \equiv D e' [\text{dom } (\text{lcl } s') ] \)
using \textit{reds}

**proof** (induct rule:converse-rtrancl-induct2)

\begin{itemize}
  \item case refl thus \(\text{?case} \).
\end{itemize}

\textbf{next}

\begin{itemize}
  \item case \((\text{step} \; e \; s \; e' \; s')\) thus \(\text{?case}
\end{itemize}

\textit{by}(cases \(s\),cases \(s'\))(auto \ dest:red-preserves-defass[OF \(\text{wf}\)])

\(\text{qed}\)

\textbf{lemma} \(\text{Red-preserves-type:}\)

\textbf{assumes} \(\text{wf: } \text{wf-\text{J-prog} } P \text{ and } \text{Red: } P \vdash \langle e,s \rangle \rightarrow \ast \langle e',s' \rangle\)

\textbf{shows} \(\Rightarrow \exists \; T'. \; P \vdash \vdash T' \leq T \wedge P, E, hp \; s' \vdash e', T'\)

\subsection*{2.22.4 Lifting to \(\Rightarrow\)}

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

\textbf{lemma} \(\text{eval-preserves-sconf:}\)

\(\llbracket \text{wf-\text{J-prog} } P; \; P \vdash \langle e,s \rangle \Rightarrow \langle e',s' \rangle; \; P, E \vdash e :: T; \; P, E \vdash s \square \rrbracket \Rightarrow P, E \vdash s' \square\)

\textbf{lemma} \(\text{eval-preserves-type: assumes } \text{wf: } \text{wf-\text{J-prog} } P\)

\textbf{shows} \(\Rightarrow \exists \; T'. \; P \vdash \vdash T' \leq T \wedge P, E, hp \; s' \vdash e', T'\)

\subsection*{2.22.5 The final polish}

The above preservation lemmas are now combined and packed nicely.

\textbf{definition} \(\text{wf-config :: } \text{J-prog } \Rightarrow \text{env } \Rightarrow \text{state } \Rightarrow \text{expr } \Rightarrow \text{ty } \Rightarrow \text{bool} \; (\ldots, \vdash \cdot \cdot \cdot \vdash \cdot \cdot \cdot \ \square \llbracket \llbracket 51, 0, 0, 0, 0, 0 \rrbracket 50\rrbracket\)

\textbf{where}

\(P, E, s \vdash e :: T \; \square \quad \equiv \quad P, E \vdash s \; \square \wedge P, E, hp \; s \vdash e :: T\)

\textbf{theorem} \(\text{Subject-reduction: assumes } \text{wf: } \text{wf-\text{J-prog} } P\)

\textbf{shows} \(P \vdash \langle e, s \rangle \rightarrow \langle e', s' \rangle \Rightarrow P, E, s \vdash e :: T \; \square \wedge P \vdash e :: T \)

\textbf{theorem} \(\text{Subject-reductions:}\)

\textbf{assumes} \(\text{wf: } \text{wf-\text{J-prog} } P \text{ and } \text{reds: } P \vdash \langle e, s \rangle \rightarrow \ast \langle e', s' \rangle\)

\textbf{shows} \(\Rightarrow \exists \; T'. \; P, E, s' \vdash e' :: T' \; \square \wedge P \vdash e' :: T' \)

\textbf{corollary} \(\text{Progress: assumes } \text{wf: } \text{wf-\text{J-prog} } P\)

\textbf{shows} \(\Rightarrow P, E, s \vdash e :: T \; \square \wedge D \; e \; \ldots :: \text{final } e \; \Rightarrow \exists \; e' s'. \; P \vdash \langle e, s \rangle \rightarrow \langle e', s' \rangle\)

\textbf{corollary} \(\text{TypeSafety:}\)

\(\Rightarrow (\exists \; a. \; e' = \text{Val } v \wedge P, hp \; s' \vdash v :: T) \vee \)

\(\Rightarrow (\exists \; a. \; e' = \text{Throw } a \wedge a \in \text{dom}(hp \; s'))\)

\textbf{end}

\section*{2.23 Program annotation}

\textbf{theory} \(\text{Annotate imports WellType begin}\)
inductive
   \[ \text{Anno} :: [\text{J-prog,env, expr , expr}] \Rightarrow \text{bool} \]
   \[ (\_, \cdot \vdash - \sim - [51,0,0,51]) \]
   \textbf{and} \[ \text{Annos} :: [\text{J-prog,env, expr list, expr list}] \Rightarrow \text{bool} \]
   \[ (\_, \cdot \vdash - [\sim] - [51,0,0,51]) \]
for \( P :: \text{J-prog} \)

where

AnnoNew: \( P,E \vdash \text{new } C \sim \text{new } C \)

| AnnoCast: \( P,E \vdash e \sim e' \Rightarrow P,E \vdash \text{Cast } C e \sim \text{Cast } C e' \)
| AnnoVal: \( P,E \vdash \text{Val } v \sim \text{Val } v \)
| AnnoVarVar: \( E V = [T] \Rightarrow P,E \vdash \text{Var } V \sim \text{Var } V \)
| AnnoVarField: \( E V = \text{None}; E \text{this} = \{\text{Class } C\}; P \vdash C \text{sees } V:T \text{ in } D \)
   \[ \Rightarrow P,E \vdash \text{Var } V \sim \text{Var this}\cdot V\{D\} \]
| AnnoBinOp:
   \[ \left[ P,E \vdash e_1 \sim e_1'; P,E \vdash e_2 \sim e_2' \right] \]
   \[ \Rightarrow P,E \vdash e_1 <\text{bop}> e_2 \sim e_1'<\text{bop}> e_2' \]
| AnnoLAssocVar: \( E V = [T]; P,E \vdash e \sim e' \) \[ \Rightarrow P,E \vdash V := e \sim V := e' \]
| AnnoLAssField: \( E V = \text{None}; E \text{this} = \{\text{Class } C\}; P \vdash C \text{sees } V:T \text{ in } D; P,E \vdash e \sim e' \)
   \[ \Rightarrow P,E \vdash V := e \sim \text{Var this}\cdot V\{D\} := e' \]
| AnnoFApply:
   \[ \left[ P,E \vdash e \sim e'; P,E \vdash e' :: \text{Class } C; P \vdash C \text{sees } F:T \text{ in } D \right] \]
   \[ \Rightarrow P,E \vdash e\cdot F\{[]\} \sim e'\cdot F\{D\} \]
| AnnoFApply:
   \[ \left[ P,E \vdash e_1 \sim e_1'; P,E \vdash e_2 \sim e_2'; P,E \vdash e_1' :: \text{Class } C; P \vdash C \text{sees } F:T \text{ in } D \right] \]
   \[ \Rightarrow P,E \vdash e_1\cdot F\{[]\} := e_2 \sim e_1'\cdot F\{D\} := e_2' \]
| AnnoCall:
   \[ \left[ P,E \vdash e \sim e'; P,E \vdash \text{es } [\sim] \text{es}' \right] \]
   \[ \Rightarrow P,E \vdash \text{Call } e \text{ } M \text{es } \sim \text{Call e'} M \text{es}' \]
| AnnoBlock:
   \[ P,E\{V \mapsto T\} \vdash e \sim e' \Rightarrow P,E \vdash \{V:T; e\} \sim \{V:T; e'\} \]
| AnnoComp:
   \[ \left[ P,E \vdash e_1 \sim e_1'; P,E \vdash e_2 \sim e_2' \right] \]
   \[ \Rightarrow P,E \vdash e_1\cdot e_2 \sim e_1'\cdot e_2' \]
| AnnoCond:
   \[ \left[ P,E \vdash e \sim e'; P,E \vdash e_1 \sim e_1'; P,E \vdash e_2 \sim e_2' \right] \]
   \[ \Rightarrow P,E \vdash \text{if (e) } e_1 \text{else } e_2 \sim \text{if (e') } e_1' \text{else } e_2' \]
| AnnoLoop:
   \[ \left[ P,E \vdash e \sim e'; P,E \vdash c \sim c' \right] \]
   \[ \Rightarrow P,E \vdash \text{while (c) } c \sim \text{while (c') } c' \]
| AnnoThrow:
   \[ P,E \vdash e \sim e' \Rightarrow P,E \vdash \text{throw } e \sim \text{throw } e' \]
| AnnoTry:
   \[ \left[ P,E \vdash e_1 \sim e_1'; P,E\{V \mapsto \text{Class } C\} \vdash e_2 \sim e_2' \right] \]
   \[ \Rightarrow P,E \vdash \text{try } e_1 \text{catch}(C V) e_2 \sim \text{try } e_1' \text{catch}(C V) e_2' \]
| AnnoNil:
   \[ P,E \vdash \cdot [\sim] \cdot \]
| AnnoCons:
   \[ \left[ P,E \vdash e \sim e'; P,E \vdash \text{es } [\sim] \text{es}' \right] \]
   \[ \Rightarrow P,E \vdash e\#\text{es } [\sim] \text{es}' \]
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],["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"]]) )
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 ⊢ ⟨es, s1⟩ ⇒ (e; vs) ; 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⟩ ⇒ (e; vs) ; 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 ⊢ ⟨addr a, s0⟩; P ⊢ ⟨ps, s1⟩ ⇒ (e; vs, (h2, l2)) ;
    map-val evs vs;
    h2 a = Some(C; js); P ⊢ C sees M:Ts⇒T = (pns; body) in D;
    length vs = length pns; l2’ = [this⇒Addr a, pns];
    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: $i \Rightarrow o \Rightarrow \text{bool}$)
map-val
.

code-pred
(modes: $i \Rightarrow o \Rightarrow o \Rightarrow \text{bool}$)
map-val2
.

lemmas [code-pred-intro] =
eval-evals.New eval-evals.NewFail
eval-evals.Val eval-evals.Var
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.CallObj

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.While eval-evals.WhileF eval-evals.WhileT
eval-evals.WhileCond

declare eval-evals.WhileBody [code-pred-intro WhileBody]

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: $i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow o \Rightarrow \text{bool as execute}$)
eval
proof –
case eval
from eval.prems show thesis
proof (cases (no-simp))
  case CallNull thus ?thesis
    by (rule eval.CallNull2 [OF refl]) (simp add: map-val-conv[symmetric])
next
case CallParamsThrow thus ?thesis
  by (rule eval.CallParamsThrow2 [OF refl]) (simp add: map-val2-conv[symmetric])
next
case Call thus ?thesis
by -(rule eval.Call2[OF refl], simp-all add: map-val-conv[ symmetric])

next

  case WhileBodyThrow thus \thesis by (rule eval.WhileBodyThrow[OF refl])

  qed | assumption | erule (4) eval.that [OF refl] | erule (3) eval.that [OF refl] +

next

  case evals

from evals.prems show \thesis

  by (cases (no-simp))(assumption | erule (3) evals.that [OF refl]) +

  qed

notation execute (- |- '(1;/)-') \Rightarrow/ \langle', '-'\rangle [51,0,0] 81

definition test1 = \[ \Rightarrow \langle testExpr1,(empty,empty) \rangle \Rightarrow \langle', '-'\rangle

definition test2 = \[ \Rightarrow \langle testExpr2,(empty,empty) \rangle \Rightarrow \langle', '-'\rangle

definition test3 = \[ \Rightarrow \langle testExpr3,(empty,empty,("V"\Rightarrow Intg 77)) \rangle \Rightarrow \langle', '-'\rangle

definition test4 = \[ \Rightarrow \langle testExpr4,(empty,empty) \rangle \Rightarrow \langle', '-'\rangle

definition test5 = \[ ("Object","","","","",[[],[]]), ("C","","","",[[],[]]) \Rightarrow \langle testExpr5,(empty,empty) \rangle \Rightarrow \langle', '-'\rangle

definition test6 = \[ ("Object",","","",[[],[]]), classL \Rightarrow \langle testExpr6,(empty,empty) \rangle \Rightarrow \langle', '-'\rangle


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] \Rightarrow (testExpr-BuildList, (empty,empty)) \Rightarrow (',-')

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} 2)) andalso
   fs2 (F, L) = SOME (@{code Intg} (@{code int-of-integer} 2)) andalso
   fs2 (N, L) = SOME (@{code Addr} (@{code nat-of-integer} 3)) andalso
   fs3 (F, L) = SOME (@{code Intg} (@{code int-of-integer} 3)) andalso
   fs3 (N, L) = SOME (@{code Addr} (@{code nat-of-integer} 4)) 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});
⟩⟩

2.26 Code Generation For WellType

theory execute-WellType
imports
   WellType Examples
begin

lemma WTCond1:
   [P, E ⊢ e :: Boolean; P, E ⊢ e₁ :: T₁; P, E ⊢ e₂ :: T₂; P ⊢ T₁ ≤ T₂;
      P ⊢ T₂ ≤ T₁ → T₂ = T₁] ⇒ P, E ⊢ if (e) e₁ else e₂ :: T₂
by (fastforce)
lemma WTCond2:
\[ \begin{align*}
P, E \vdash e :: \text{Boolean}; & \quad P, E \vdash e_1 :: T_1; \quad P, E \vdash e_2 :: T_2; \quad P \vdash T_2 \leq T_1; \\
P \vdash T_1 \leq T_2 \quad \rightarrow \quad T_1 = T_2 \quad \implies \quad P, E \vdash \text{if } (e) \ e_1 \ \text{else } e_2 :: T_1 \end{align*} \]

by (fastforce)

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

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

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

code-pred
(modes: i ⇒ i ⇒ i ⇒ bool as type-check, i ⇒ i ⇒ o ⇒ bool as infer-type)

proof -
  case WT
  from WT.prems show thesis

proof (cases (no-simp))
  case (WTCond E e e1 T1 e2 T2 T)
  from \( (x \vdash T_1 \leq T_2 \lor x \vdash T_2 \leq T_1) \) show thesis

proof
  assume \( x \vdash T_1 \leq T_2 \)
  with \( x \vdash T_1 \leq T_2 \quad \rightarrow \quad T = T_2 \) have \( T = T_2 \) ..

  from \( \langle xa = E \rangle \langle xb = \text{if } (e) \ e_1 \ \text{else } e_2 \rangle \langle xc = T \rangle \langle x, E \vdash e :: \text{Boolean} \rangle \langle x, E \vdash e_1 :: T_1 \rangle \langle x, E \vdash e_2 :: T_2 \rangle \langle x \vdash T_1 \leq T_2 \rangle \langle x \vdash T_2 \leq T_1 \quad \rightarrow \quad T = T_1 \rangle \)

  show ?thesis unfolding \( \langle T = T_2 \rangle \) by (rule WT.WTCond1[OF refl])

next
  assume \( x \vdash T_2 \leq T_1 \)
  with \( x \vdash T_2 \leq T_1 \quad \rightarrow \quad T = T_1 \) have \( T = T_1 \) ..

  from \( \langle xa = E \rangle \langle xb = \text{if } (e) \ e_1 \ \text{else } e_2 \rangle \langle xc = T \rangle \langle x, E \vdash e :: \text{Boolean} \rangle \langle x, E \vdash e_1 :: T_1 \rangle \langle x, E \vdash e_2 :: T_2 \rangle \langle x \vdash T_2 \leq T_1 \rangle \langle x \vdash T_1 \leq T_2 \quad \rightarrow \quad T = T_2 \rangle \)

  show ?thesis unfolding \( \langle T = T_1 \rangle \) by (rule WT.WTCond2[OF refl])

qed

qed (assumption | erule (2) WT.that[OF refl])+
next
case WTs
from WTs.prems show thesis
  by (cases (no-simp))(assumption|erule (2) WTs.that[OF refl])+ qed

notation infer-type (-,- |- - |- - |- (51,51,51) 100)
definition test1 where test1 = [], 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", (["F", Integer]|[], [])]), empty |- testExpr5 :: -
definition test6 where test6 = [classObject, classI], empty |- testExpr6 :: -

ML-val ""
val SOME(lookup code Integer, -) = Predicate.yield lookup code test1;
val SOME(lookup code Integer, -) = Predicate.yield lookup code test2;
val SOME(lookup code Integer, -) = Predicate.yield lookup code test3;
val SOME(lookup code Void, -) = Predicate.yield lookup code test4;
val SOME(lookup code Void, -) = Predicate.yield lookup code test5;
val SOME(lookup code Integer, -) = Predicate.yield lookup 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]) |- mb-mult-block :: -
definition testmb-mult where testmb-mult = [classObject, classA], empty([this,"i","j","temp"]
  |- [Class "A", Integer, Integer]) |- mb-mult :: -

ML-val ""
val SOME(lookup code Boolean, -) = Predicate.yield lookup code testmb-isNull;
val SOME(lookup code Integer, -) = Predicate.yield lookup code testmb-add;
val SOME(lookup code Boolean, -) = Predicate.yield lookup code testmb-mult-cond;
val SOME(lookup code Integer, -) = Predicate.yield lookup code testmb-mult-block;
val SOME(lookup code Integer, -) = Predicate.yield lookup code testmb-mult;"

definition test where test = [classObject, classA], empty |- testExpr-ClassA :: -

ML-val ""
val SOME(lookup code Integer, -) = Predicate.yield lookup 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

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

end

3.2 Instructions of the JVM

theory JVMInstructions imports JVMState begin

end
| 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, (↓il 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-syst 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-syst 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-syst NullPointer] else None;
   a = the-Addr r;
   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-syst-xt CastCast] 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 = stkn;
   xp' = if r=Null then [addr-of-syst NullPointer] else None;
   C = fst((h(the-Addr r)));
   (D,M',Ts,mzs,mzl,ins,xt)= method P C M;
   f' = ([[],[r]@[rev ps]@[replicate mzl undefined],D,M,0)
   in (xp', h, f'#(stk, loc, C₀, M₀, pc)#frs))

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

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

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

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

| exec-instr CmpEq P h stk loc C₀ M₀ pc frs =
\[
\begin{align*}
&\text{lemma exec-instr-Return:} \\
&\quad \text{exec-instr Return} P h \ (v\#stk) \ loc C_0 \ M_0 \ pc \ frs = \\
&\quad \begin{cases}
&\text{let} \ v = \text{hd} \ (\text{tl} \ stk) \\
&\text{in} \ (\text{None}, h, (\text{Bool} \ (v = v_2) \ # \ \text{tl} \ (\text{tl} \ stk), \ loc, C_0, M_0, pc+1)\#frs))
&\end{cases}
\end{align*}
\]

\[
| \text{exec-instr-Goto:} \\
|\text{exec-instr} \ (\text{Goto} \ i) \ P h \ stk \ loc \ C_0 \ M_0 \ pc \ frs = \\
| \quad (\text{None}, h, (\text{stk}, \ loc, C_0, M_0, \ \text{nat}(\text{int} \ pc+i))\#frs)
\]

\[
| \text{exec-instr Throw} P h \ stk \ loc \ C_0 \ M_0 \ pc \ frs = \\
| \quad (\text{let} \ xp' = \text{if} \ h = \text{Null} \ \text{then} \ [\text{addr-of-sys-xcpt} \ \text{NullPointerException}] \ \text{else} \ [\text{the-Addr}(h \ \text{stk})] \\
|\quad \text{in} \ (xp', h, (\text{stk}, \ loc, C_0, M_0, pc)\#frs))
\]

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

\[
\text{lemma exec-instr-Getfield:} \\
\text{exec-instr} \ (\text{Getfield} \ F \ C) \ P h \ (v\#stk) \ loc C_0 \ M_0 \ pc \ frs = \\
\quad (\text{let} \ xp' = \text{if} \ v = \text{Null} \ \text{then} \ [\text{addr-of-sys-xcpt} \ \text{NullPointerException}] \ \text{else} \ None; \\
\quad \quad (D, fs) = \text{the}(h(\text{the-Addr} v)); \\
\quad \quad h' = h(a \mapsto (D, fs((F, C) \mapsto v))) \\
\quad \text{in} \ (xp', h', (\text{stk}, \ loc, C_0, M_0, pc+1)\#frs)) \\
\quad \text{by simp}
\]

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

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

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

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

\[
\text{lemma exec-instr-IAdd:}
\]
Theory JVMExceptions

exec-instr $IAdd\ P\ h\ (Intg\ i_2\ #\ Intg\ i_1\ #\ stk)\ loc\ C_0\ M_0\ pc\ frs =$
\quad (None,\ h,\ (Intg\ (i_1+i_2)\ #\ stk,\ loc,\ C_0,\ M_0,\ pc+1)\ #\ frs)$
by simp

lemma exec-instr-IfFalse:
exec-instr $(IfFalse\ i)\ P\ h\ (v\ #\ stk)\ loc\ C_0\ M_0\ pc\ frs =$
\quad (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 $CmpEq\ P\ h\ (v_2\ #\ v_1\ #\ stk)\ loc\ C_0\ M_0\ pc\ frs =$
\quad (None,\ h,\ (Bool\ (v_1\ =\ v_2)\ #\ stk,\ loc,\ C_0,\ M_0,\ pc+1)\ #\ frs)$
by simp

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

end

3.4 Exception handling in the JVM

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

definition matches-ex-entry :: 'm prog ⇒ cname ⇒ pc ⇒ ex-entry ⇒ bool
where
matches-ex-entry $P\ C\ pc\ xcp ≡$
\quad let\ (s,\ e,\ C',\ h,\ d) = xcp in
\quad s ≤ pc ∧ pc < e ∧ P ⊩ C' ≤∗ C'$

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

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

primrec find-handler :: jvm-prog ⇒ addr ⇒ heap ⇒ frame list ⇒ jvm-state
where
find-handler $P\ a\ h\ [] = (Some\ a,\ h,\ [])$
\ | find-handler $P\ a\ h\ (fr#frs) =$
\quad (let\ (stk,loc,C,M,pc) = fr\ in
\quad case\ match-ex-table\ P\ (cname-of\ h\ a)\ pc\ (ex-table-of\ P\ C\ M)\ of
\quad 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))

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(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−→/ − [61,61,61] 60)
for P :: jvm-prog
where
P ⊢ σ − jvm−→₁ σ' ≡ (σ,σ') ∈ exec-1 P
| exec-11: 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-1-1:
  (σ,σ'). exec (P,σ) = Some σ'
lemma exec-1-1-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')*)
The start configuration of the JVM: in the start heap, we call a method \( m \) of class \( C \) in program \( P \). The \texttt{this} pointer of the frame is set to \texttt{Null} to simulate a static method invocation.

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

### 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 `'

**Function**

- \texttt{is-Addr :: val ⇒ bool} where
  \[
  \texttt{is-Addr (Addr a) } ←→ \texttt{True} \\
  \texttt{is-Addr v } ←→ \texttt{False}
  \]

- \texttt{is-Intg :: val ⇒ bool} where
  \[
  \texttt{is-Intg (Intg i) } ←→ \texttt{True} \\
  \texttt{is-Intg v } ←→ \texttt{False}
  \]

- \texttt{is-Bool :: val ⇒ bool} where
  \[
  \texttt{is-Bool (Bool b) } ←→ \texttt{True} \\
  \texttt{is-Bool v } ←→ \texttt{False}
  \]

**Definition** \texttt{is-Ref :: val ⇒ bool} where
\[
\texttt{is-Ref v} ←→ v = \texttt{Null} \lor \texttt{is-Addr v}
\]

**Primrec** \texttt{check-instr :: [instr, jvm-prog, heap, val list, val list, cname, mname, pc, frame list] ⇒ bool} where

- \texttt{check-instr (Load n) P h stk loc C M_0 pc frs} =
(n < length loc)

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

| check-instr-Push:  
| check-instr (Push v) P h stk loc C_0 M_0 pc frs =  
| (∼is-Addr v)

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

| check-instr-Getfield:  
| check-instr (Getfield F C) P h stk loc C_0 M_0 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 C_0 M_0 pc frs =  
| (1 < length stk ∧ (∃ C' T. P ⊢ C sees F:T in C') ∧  
| (let (C', T) = field P C F; v = hd stk; ref = hd (tl stk) in  
| C' = C ∧ is-Ref ref ∧ (ref ≠ Null →  
| h (the-Addr ref) ≠ None ∧  
| (let D = fst (the (h (the-Addr ref))) in  
| P ⊢ D ≤^∗ C ∧ P,h ⊢ v :≤ T))))

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

| check-instr-Invoke:  
| check-instr (Invoke M n) P h stk loc C_0 M_0 pc frs =  
| (n < length stk ∧ is-Ref (stk!n) ∧  
| (stk!n ≠ Null →  
| (let a = the-Addr (stk!n);  
| C = cname-of h a;  
| Ts = fst (snd (method P C M))  
| in h a ≠ None ∧ P ⊢ C has M ∧  
| P,h ⊢ rev (take n stk) [:≤] Ts)))

| check-instr-Return:  
| check-instr Return P h stk loc C_0 M_0 pc frs =  
| (0 < length stk ∧ (0 < length frs) →  
| (P ⊢ C_0 has M_0) ∧  
| (let v = hd stk;  
| T = fst (snd (snd (method P C_0 M_0))))  
| in P,h ⊢ v :≤ T)))
theory JVMDefensive

<table>
<thead>
<tr>
<th>check-instr-Pop:</th>
</tr>
</thead>
<tbody>
<tr>
<td>check-instr Pop P h stk loc C₀ M₀ pc frs =</td>
</tr>
<tr>
<td>(0 &lt; length stk)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>check-instr-IAdd:</th>
</tr>
</thead>
<tbody>
<tr>
<td>check-instr IAdd P h stk loc C₀ M₀ pc frs =</td>
</tr>
<tr>
<td>(1 &lt; length stk ∧ is-Intg (hd stk) ∧ is-Intg (hd (tl stk)))</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>check-instr-IfFalse:</th>
</tr>
</thead>
<tbody>
<tr>
<td>check-instr (IfFalse b) P h stk loc C₀ M₀ pc frs =</td>
</tr>
<tr>
<td>(0 &lt; length stk ∧ is-Bool (hd stk) ∧ 0 ≤ int pc+b)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>check-instr-CmpEq:</th>
</tr>
</thead>
<tbody>
<tr>
<td>check-instr CmpEq P h stk loc C₀ M₀ pc frs =</td>
</tr>
<tr>
<td>(1 &lt; length stk)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>check-instr-Goto:</th>
</tr>
</thead>
<tbody>
<tr>
<td>check-instr (Goto b) P h stk loc C₀ M₀ pc frs =</td>
</tr>
<tr>
<td>(0 ≤ int pc+b)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>check-instr-Throw:</th>
</tr>
</thead>
<tbody>
<tr>
<td>check-instr Throw P h stk loc C₀ M₀ pc frs =</td>
</tr>
<tr>
<td>(0 &lt; length stk ∧ is-Ref (hd stk))</td>
</tr>
</tbody>
</table>

definition check :: jvm-prog ⇒ jvm-state ⇒ bool where
check P σ = (let (xcpt, h, frs) = σ in
(case frs of [] ⇒ True | (stk,loc,C,M,pc)#frs' ⇒
P ⊢ C has M ∧
(let (C',Ts,T,maxs,mxl,ins,xt) = method P C M; i = ins!pc in
pc < size ins ∧ size stk ≤ maxs ∧
check-instr i P h stk loc C M pc frs'))) |

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

inductive-set
exec-1-d :: jvm-prog ⇒ (jvm-state type-error × jvm-state type-error) set
and exec-1-d' :: jvm-prog ⇒ jvm-state type-error ⇒ jvm-state type-error ⇒ bool
(- ⊢ - -jvmd→₁ - [61,61,61][60])

for P :: jvm-prog
where
P ⊢ σ -jvmd→₁ σ' ≡ (σ,σ') ∈ exec-1-d P

| exec-1-d-ErrorI: exec-d P σ = TypeError ⇒ P ⊢ Normal σ -jvmd→₁ TypeError |
| exec-1-d-NormalI: exec-d P σ = Normal (Some σ') ⇒ P ⊢ Normal σ -jvmd→₁ Normal σ' |

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

notation (xsymbols)
exect-all-d (- ⊢ - -jvmd→ - [61,61,61][60])
lemma exec-1-d-eq:
    exec-1-d P = \{(s,t). \exists \sigma. s = \text{Normal } \sigma \land t = \text{TypeError } \land \text{exec-d } P \sigma = \text{TypeError}\} \cup
    \{(s,t). \exists \sigma'. s = \text{Normal } \sigma \land t = \text{Normal } \sigma' \land \text{exec-d } P \sigma = \text{Normal } (\text{Some } \sigma')\}
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 \implies P
by (cases P, auto)

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

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

lemma defensive-imp-aggressive:
    P \vdash (\text{Normal } \sigma) \rightarrow \text{jvm} \rightarrow (\text{Normal } \sigma') \implies P \vdash \sigma \rightarrow \text{jvm} \rightarrow \sigma'
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, NullPointerException, 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,
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
defclare undefined-cname-def [symmetric, code-unfold]
code-printing constant undefined-cname --> (SML) object

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

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

definition test = exec (E, start-state E test-name makelist-name)
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;
80

} end
Chapter 4

Bytecode Verifier

4.1 Semilattices
	heory 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
    lessub :: 'a ⇒ 'a ord ⇒ 'a ⇒ bool
    plussub :: 'a ⇒ ('a ⇒ 'b ⇒ 'c) ⇒ 'b ⇒ 'e

  notation (xsymbols)
    lesub ((- /∈. -) [50, 0, 51] 50)
    lessub ((- /⊂. -) [50, 0, 51] 50)
    plussub ((- /⊔. -) [65, 0, 66] 65)

 defs
    lesub-def : x ⊑ r y ≡ r x y
    lessub-def : x ⊏ r y ≡ x ⊑ r y ∧ x ≠ y
    plussub-def : x ⊔ f y ≡ f x y

  definition ord :: ('a × 'a) set ⇒ 'a ord
       where
             ord r = (λx y. (x,y) ∈ r)

  definition order :: 'a ord ⇒ bool
       where
             order r = (∀ x. x ∈ r x) ∧ (∀ x y. x ∈ r y ∧ y ∈ r x → x=y) ∧ (∀ x y z. x ∈ r y ∧ y ∈ r z → x ∈ r z)

  definition top :: 'a ord ⇒ 'a ⇒ bool
       where
             top r T = (∀ x. x ∈ r T)

  definition acc :: 'a ord ⇒ bool
       where
             acc r = wf {(y,x). x ⊑ r y}
definition closed :: 'a set ⇒ 'a binop ⇒ bool
where
closed A f ←→ (∀ x ∈ A. ∀ y ∈ A. x ⊑ f y ∈ A)

definition semilat :: 'a sl ⇒ bool
where
semilat = (λ(A,r,f). order r ∧ closed A f ∧
(∀ x ∈ A. ∀ y ∈ A. x ⊑ r x ⊑ f y) ∧
(∀ x ∈ A. ∀ y ∈ A. y ⊑ r x ⊑ f y) ∧
(∀ x ∈ A. ∀ y ∈ A. ∀ z ∈ A. x ⊑ r z ∧ y ⊑ r x → x ⊑ f y ⊑ r z))

definition is-lub :: ('a × 'a) set ⇒ 'a ⇒ 'a ⇒ bool
where
is-lub r x y u ←→ (x,u) ∈ r ∧ (y,u) ∈ r

definition is-ub :: ('a × 'a) set ⇒ 'a ⇒ 'a ⇒ bool
where
is-ub r x y u ←→ is-lub r x y u ∧ (∀ z. is-ub r x y z → (u,z) ∈ r)

definition some-lub :: ('a × 'a) set ⇒ 'a ⇒ 'a ⇒ bool
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 ⇒ x ⊑ r x

lemma order-antisym: [ order r; x ⊑ r y; y ⊑ r x ] ⇒ x = y

lemma order-trans: [ order r; x ⊑ r y; y ⊑ r z ] ⇒ x ⊑ r z

lemma order-less-irrefl [intro, simp]: order r ⇒ ¬ x ⊑ r x

lemma order-less-trans: [ order r; x ⊑ r y; y ⊑ r z ] ⇒ x ⊑ r z

lemma topD [simp, intro]: top r T ⇒ x ⊑ r T

lemma top-le-conv [simp]: [ order r; top r T ] ⇒ (T ⊑ r x) = (x = T)

lemma semilat-Def:
semilat(A,r,f) ←→ order r ∧ closed A f ∧
(∀ x ∈ A. ∀ y ∈ A. x ⊑ r x ⊑ f y) ∧
(∀ x ∈ A. ∀ y ∈ A. y ⊑ r x ⊑ f y) ∧
(∀ x ∈ A. ∀ y ∈ A. ∀ z ∈ A. x ⊑ r z ∧ y ⊑ r x → x ⊑ f y ⊑ 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 ∈ A; y ∈ A ] ⇒ x ⊑ f y ∈ A
lemma closed-UNIV [simp]: closed UNIV \( f \)

lemma (in Semilat) closed-f [simp, intro]: \([x \in A; y \in A] \implies x \cup_f y \in A\)

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

lemma (in Semilat) antisym-r [intro?]: \([x \subseteq_r y; y \subseteq_r x] \implies x = y\)

lemma (in Semilat) trans-r [trans, intro?]: \([x \subseteq_r y; y \subseteq_r z] \implies x \subseteq_r z\)

lemma (in Semilat) \( \text{ub1} \) [simp, intro?]: \([x \in A; y \in A] \implies x \subseteq_r x \cup_f y\)

lemma (in Semilat) \( \text{ub2} \) [simp, intro?]: \([x \in A; y \in A] \implies y \subseteq_r x \cup_f y\)

lemma (in Semilat) \( \text{lub} \) [simp, intro?]:
\[
\left(\begin{array}{l}
x \subseteq_r z; \ y \subseteq_r z; x \in A; \ y \in A; z \in A
\end{array}\right) \implies x \cup_f y \subseteq_r z
\]

lemma (in Semilat) plus-le-conv [simp]:
\[
\left(\begin{array}{l}
x \in A; \ y \in A; z \in A
\end{array}\right) \implies (x \cup_f y \subseteq_r z) = (x \subseteq_r z \land y \subseteq_r z)
\]

lemma (in Semilat) le-iff-plus-unchanged:
assumes \( x \in A \) and \( y \in A \)
shows \( x \subseteq_r y \iff x \cup_f y = y \) (is \(?P \iff \(?Q)\)

lemma (in Semilat) le-iff-plus-unchanged2:
assumes \( x \in A \) and \( y \in A \)
shows \( x \subseteq_r y \iff x \cup_f x = y \) (is \(?P \iff \(?Q)\)

lemma (in Semilat) plus-assoc [simp]:
assumes \( a: a \in A \) and \( b: b \in A \) and \( c: c \in A \)
shows \( a \cup_f (b \cup_f c) = a \cup_f b \cup_f c \)

lemma (in Semilat) plus-com-lemma:
\( [a \in A; b \in A] \implies a \cup_f b \subseteq_r b \cup_f a \)

lemma (in Semilat) plus-commutative:
\( [a \in A; b \in A] \implies a \cup_f b = b \cup_f a \)

lemma is-labD:
is-lab \( r x y u \implies is-ub \ r x y u \land (\forall z. is-ub \ r x y z \implies (u,z) \in r)\)

lemma is-ubI:
\( [(x,u) \in r; (y,u) \in r] \implies is-ub \ r x y u\)

lemma is-ubD:
is-ub \( r x y u \implies (x,u) \in r \land (y,u) \in r\)

lemma is-lab-bigger1 [iff]:
is-lab \( (r^*) x y y = ((x,y)\in r^*) \)

lemma is-lab-bigger2 [iff]:
is-lab \( (r^*) x y x = ((y,x)\in r^*) \)

lemma extend-lab:
\[
[\text{single-valued} \ r; \text{is-lab} \ (r^*) x y u; (x',x) \in r] \\
\implies EX v. \text{is-lab} \ (r^*) x' y v
\]

lemma single-valued-has-lubs [rule-format]:
\[
[\text{single-valued} \ r; (x,u) \in r^*] \implies (\forall y. (y,u) \in r^* \implies \ldots)
\]
\((EX z. \text{is-lub} (r^{\ast \ast}) \ x \ y \ z)\)

**Lemma some-lub-conv:**
\[\left[\text{acyclic } r; \text{is-lub} (r^{\ast \ast}) \ x \ y \ u\right] \Rightarrow \text{some-lub} (r^{\ast \ast}) \ x \ y = u\]

**Lemma is-lub-some-lub:**
\[\left[\text{acyclic } r; \text{single-valued } r; (x,u) \in r^{\ast \ast}; (y,u) \in r^{\ast \ast}\right] \Rightarrow \text{is-lub} (r^{\ast \ast}) \ x \ y = (\text{some-lub} (r^{\ast \ast}) \ x \ y)\]

### 4.1.1 An executable lub-finder

**Definition** \(\text{exec-lub} :: ('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'a\)

where
\[\text{exec-lub } r \ f \ x \ y = \text{while } (\lambda z. (x,z) \notin r^{\ast}) f y\]

**Lemma exec-lub-refl:** 
\[\text{exec-lub } r \ f \ T \ T = T\]
by\((\text{simp add: exec-lub-def while-unfold})\)

**Lemma acyclic-single-valued-finite:**
\[\left[\text{acyclic } r; \text{single-valued } r; (x,y) \in r^{\ast}\right] \Rightarrow \text{finite } (r \cap \{a. (x,a) \in r^{\ast}\} \times \{b. (b,y) \in r^{\ast}\})\]

**Lemma exec-lub-conv:**
\[\left[\text{acyclic } r; \forall x y. (x,y) \in r \Rightarrow f x = y; \text{is-lub} (r^{\ast \ast}) \ x \ y \ u\right] \Rightarrow \text{exec-lub } r \ f \ x \ y = u\]

**Lemma is-lub-exec-lub:**
\[\left[\text{single-valued } r; \text{acyclic } r; (x,u) \in r^{\ast \ast}; (y,u) \in r^{\ast \ast}; \forall x y. (x,y) \in r \Rightarrow f x = y\right] \Rightarrow \text{is-lub} (r^{\ast \ast}) \ x \ y = (\text{exec-lub } r \ f \ x \ y)\]

end

### 4.2 The Error Type

**Theory** \(\text{Err}\)

**Imports** \(\text{Semilat}\)

begin

datatype 'a err = \text{Err} \mid \text{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-val :: 'a err \Rightarrow 'a
where
\[\text{ok-val } (\text{OK } x) = x\]

**Definition** \(\text{lift} :: ('a \Rightarrow 'b err) \Rightarrow ('a err \Rightarrow 'b err)\)
where
\[\text{lift } f \ e = (\text{case } e \text{ of } \text{Err} \Rightarrow \text{Err} \mid \text{OK } x \Rightarrow f x)\]

**Definition** \(\text{lift2} :: ('a \Rightarrow 'b \Rightarrow 'c err) \Rightarrow ('a err \Rightarrow 'b err \Rightarrow 'c err)\)
where
\[\text{lift2 } f \ e_1 \ e_2 = (\text{case } e_1 \text{ of } \text{Err} \Rightarrow \text{Err} \mid \text{OK } x \Rightarrow (\text{case } e_2 \text{ of } \text{Err} \Rightarrow \text{Err} \mid \text{OK } y \Rightarrow f x y))\]
**Theory Err**

**Definition** \( \text{le} :: 'a \text{ ord} \Rightarrow 'a \text{ err} \text{ ord} \)**

where

\[
\text{le } r \ e_1 \ e_2 = \\
\begin{cases}
\text{True} & \text{if } (\text{case } e_2 \text{ of } \text{Err} \Rightarrow \text{True}) \\
(\text{case } e_1 \text{ of } \text{Err} \Rightarrow \text{False}) & \text{if } \text{OK } x \Rightarrow x \subseteq_r y) \\
\end{cases}
\]

**Definition** \( \text{sup} :: ('a \Rightarrow 'b \Rightarrow 'c) \Rightarrow ('a \text{ err} \Rightarrow 'b \text{ err} \Rightarrow 'c \text{ err}) \)**

where

\[
\text{sup } f = \text{lift2 } (\lambda x \ y. \ \text{OK } (x \sqcup_f y))
\]

**Definition** \( \text{err} :: 'a \text{ set} \Rightarrow 'a \text{ err set} \)**

where

\[
\text{err } A = \text{insert } \text{Err} \ \{ \text{OK } x | x \in A \}
\]

**Definition** \( \text{esl} :: 'a \text{ sl} \Rightarrow 'a \text{ esl} \)**

where

\[
\text{esl } = (\lambda (A, r, f). \ (A, r, \lambda x \ y. \ \text{OK } (f x y)))
\]

**Definition** \( \text{sl} :: 'a \text{ esl} \Rightarrow 'a \text{ err sl} \)**

where

\[
\text{sl } = (\lambda (A, r, f). \ (\text{err } A, \ \text{le } r, \ \text{lift2 } f))
\]

**Abbreviation**

\( \text{err-semilat} :: 'a \text{ esl} \Rightarrow \text{bool} \)**

where

\[
\text{err-semilat } L = \text{semilat}(\text{sl } L)
\]

**Primrec** \( \text{strict} :: ('a \Rightarrow 'b \text{ err}) \Rightarrow ('a \text{ err} \Rightarrow 'b \text{ err}) \)**

where

\[
\text{strict } f \ \text{Err} = \text{Err} \\
\text{strict } f \ (\text{OK } x) = f \ x
\]

**Lemma** \( \text{err-def} \)**

\[
\text{err } A = \text{insert } \text{Err } \{ x. \ \exists y \in A. \ x = \text{OK } y \}
\]

**Lemma** \( \text{strict-Some} \)**

\[
(\text{strict } f \ x = \text{OK } y) = (\exists z. \ x = \text{OK } z \land f z = \text{OK } y)
\]

**Lemma** \( \text{not-Err-eq} \)**

\[
(\text{x } \neq \text{Err}) = (\exists a. \ x = \text{OK } a)
\]

**Lemma** \( \text{not-OK-eq} \)**

\[
(\forall y. \ x \neq \text{OK } y) = (x = \text{Err})
\]

**Lemma** \( \text{unfold-lesub-err} \)**

\[
\text{le } r \ e_1 \ e_2 = \text{le } r \ e_1 \ e_2
\]

**Lemma** \( \text{le-err-refl} \)**

\[ \forall x. \ x \subseteq_r x \Rightarrow e \subseteq_r e \]

**Lemma** \( \text{le-err-trans} \)**

\[ \text{order } r \Rightarrow e_1 \subseteq_r e_2 \Rightarrow e_2 \subseteq_r e_3 \Rightarrow e_1 \subseteq_r e_3 \]

**Lemma** \( \text{le-err-antisym} \)**

\[ \text{order } r \Rightarrow e_1 \subseteq_r e_2 \Rightarrow e_2 \subseteq_r e_1 \Rightarrow e_1 = e_2 \]

**Lemma** \( \text{OK-le-err-OK} \)**

\[ (\text{OK } x \subseteq_r \text{OK } y) = (x \subseteq r y) \]

**Lemma** \( \text{order-le-err} \)**

\[
\text{order}(\text{le } r) = \text{order } r
\]

**Lemma** \( \text{le-Err} \)**

\[
\text{e } \subseteq_r \text{le } r \ \text{Err}
\]

**Lemma** \( \text{Err-le-conv} \)**

\[ \text{Err } \subseteq_r \text{le } r \ \text{e } = (e = \text{Err}) \]

**Lemma** \( \text{le-OK-conv} \)**

\[
\text{e } \subseteq_r \text{le } r \ \text{OK } x = (\exists y. \ e = \text{OK } y \land y \subseteq_r x)
\]

**Lemma** \( \text{OK-le-conv} \)**

\[
\text{OK } x \subseteq_r \text{le } r \ \text{e } = (e = \text{Err} \lor (\exists y. \ e = \text{OK } y \land x \subseteq_r y))
\]

**Lemma** \( \text{top-Err} \)**

\[
\text{top } (\text{le } r) \ \text{Err}
\]

**Lemma** \( \text{OK-less-conv} \)**

\[
\text{OK } x \subseteq_r \text{le } r \ \text{e } = (e = \text{Err} \lor (\exists y. \ e = \text{OK } y \land x \subseteq_r y))
\]

**Lemma** \( \text{not-Err-less} \)**

\[
(\neg \text{Err } \subseteq_r x)
\]

**Lemma** \( \text{semilat-errI} \)**

\[
\text{Semilat } A \ r f
\]

**Shows** \( \text{semilat}(\text{err } A, \text{le } r, \text{lift2 } (\lambda x \ y. \ \text{OK } (f x y))) \)
lemma err-semilat-eslI-aux:
assumes Semilat \( A \ r \ f \) shows err-semilat(esl\((A,r,f)\))

lemma err-semilat-eslI [intro, simp]:
\( \text{semilat } L \implies \text{err-semilat } (\text{esl } L) \)

lemma acc-err [simp, intro!]: \( \text{acc } r \implies \text{acc}(le r) \)

lemma Err-in-err [iff]: \( \text{Err} : \text{err } A \)

lemma Ok-in-err [simp]: \( \text{OK } x \in \text{err } A = (x \in A) \)

4.2.1 lift

lemma lift-in-err1:
\[ e \in \text{err } S; \forall x \in S, e = \text{OK } x \rightarrow f x \in \text{err } S \implies \text{lift } f e \in \text{err } S \]

lemma Err-lift2 [simp]: \( \text{Err } \cup_{\text{lift2}} f x = \text{Err} \)

lemma lift2-Err [simp]: \( x \cup_{\text{lift2}} f \text{Err} = \text{Err} \)

lemma OK-lift2-OK [simp]: \( \text{OK } x \cup_{\text{lift2}} f \text{OK } y = x \cup f y \)

4.2.2 sup

lemma Err-sup-Err [simp]: \( \text{Err } \cup_{\text{sup}} f x = \text{Err} \)

lemma Err-sup-Err2 [simp]: \( x \cup_{\text{sup}} f \text{Err} = \text{Err} \)

lemma Err-sup-OK [simp]: \( \text{OK } x \cup_{\text{sup}} f \text{OK } y = \text{OK } (x \cup f y) \)

lemma Err-sup-OK-lift2-OK-conv [simp]:
\( (\text{sup } f ex ey = \text{OK } z) = (\exists x y, ex = \text{OK } x \land ey = \text{OK } y \land f x y = z) \)

lemma Err-sup-OK-lift2-Err-conv [iff]: \( (\text{sup } f ex ey = \text{Err}) = (ex=\text{Err} \lor ey=\text{Err}) \)

4.2.3 semilat (\( \text{err } A \)) (le r) f

lemma semilat-le-err-Err-plus [simp]:
\[ x \in \text{err } A; \text{semilat}(\text{err } A, \text{le } r, f) \implies \text{Err } \cup_{f} x = \text{Err} \]

lemma semilat-le-err-plus-Err [simp]:
\[ x \in \text{err } A; \text{semilat}(\text{err } A, \text{le } r, f) \implies x \cup_{f} \text{Err} = \text{Err} \]

lemma semilat-le-err-OK1:
\[ x \in A; y \in A; \text{semilat}(\text{err } A, \text{le } r, f); \text{OK } x \cup_{f} \text{OK } y = \text{OK } z \]
\[ \implies x \sqsubseteq_{r} z \]

lemma semilat-le-err-OK2:
\[ x \in A; y \in A; \text{semilat}(\text{err } A, \text{le } r, f); \text{OK } x \cup_{f} \text{OK } y = \text{OK } z \]
\[ \implies y \sqsubseteq_{r} z \]

lemma eq-order-le:
\[ x=y; \text{order } r \implies x \sqsubseteq_{r} y \]

lemma OK-plus-OK-err-conv [simp]:
assumes \( x \in A \) \( y \in A \) \( \text{semilat}(\text{err } A, \text{le } r, fe) \)
shows \( \text{OK } x \cup_{f} \text{OK } y = \text{Err} = (\neg(\exists x \in A. x \sqsubseteq_{r} z \land y \sqsubseteq_{r} z)) \)

4.2.4 semilat (err(\text{Union } AS))

lemma all-bex-swap-lemma [iff]:
\( (\forall x, (\exists y \in A. x = f y) \implies P x) = (\forall y \in A. P(f y)) \)

lemma closed-err-Union-lift2:
\[ (\forall A \in AS. \text{closed } (\text{err } A) (\text{lift2 } f); \ AS \neq \{}; \quad \forall A C A \in AS B \in AS. A \neq B \rightarrow (\forall a \in A \forall b \in B. a \cup f b = \text{Err}) \]
\[ \implies \text{closed } (\text{err}(\text{Union } AS)) (\text{lift2 } f) \]

If \( AS = \{} \) the thm collapses to \text{order } r \land \text{closed } \{ \text{Err} \} f \land \text{Err } \cup_{f} \text{Err} = \text{Err} which may not hold
lemma err-semilat-UnionI:
\[
\forall A \in AS. \ err-semilat(A, r, f); \ AS \neq \{\}
\quad \forall A \in AS. \forall B \in AS. \ A \neq B \rightarrow (\forall a \in A. \forall b \in B. \ \neg a \sqsubseteq_r b \land a \sqcup_f b = \text{Err}) \\
\rightarrow err-semilat(\text{Union } AS, r, f)
\]
end

4.3 More about Options

theory Opt imports Err begin

definition le :: 'a ord ⇒ 'a option ord
where
le r o₁ o₂ = 
(case o₂ of None ⇒ o₁=None | Some y ⇒ (case o₁ 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 o₁ o₂ = 
(case o₁ of None ⇒ OK o₂ \\
Some x ⇒ (case o₂ of None ⇒ OK o₁ \\
Some y ⇒ (case f x y of Err ⇒ Err | OK z ⇒ OK (Some z))))

definition esl :: 'a esl ⇒ 'a option esl
where
esl = (\lambda (A,r,f). (opt A, le r, sup f))

lemma unfold-le-opt:
o₁ \sqsubseteq_{le r} o₂ = 
(case o₂ of None ⇒ o₁=None | 
Some y ⇒ (case o₁ of None ⇒ True | Some x ⇒ x \sqsubseteq_r y))

lemma le-opt-refl: order r \rightarrow x \sqsubseteq_{le r} x

4.4 Products as Semilattices

theory Product imports Err begin

definition le :: 'a ord ⇒ 'b ord ⇒ ('a × 'b) ord
where
le r₁ r₂ = (\lambda(a₁,b₁) (a₂,b₂). a₁ \sqsubseteq_{r₁} a₂ \land b₁ \sqsubseteq_{r₂} b₂)

definition sup :: 'a ebinop ⇒ 'b ebinop ⇒ ('a × 'b) ebinop
where
sup f g = (\lambda(a₁,b₁)(a₂,b₂). Err.sup Pair (a₁ \sqcup_f a₂) (b₁ \sqcup_g b₂))

definition esl :: 'a esl ⇒ 'b esl ⇒ ('a × 'b) esl
where
\[ \text{esl} = (\lambda (A, r_A, f_A) \ (B, r_B, f_B). \ (A \times B, \ le r_A r_B, \ sup f_A f_B)) \]

**abbreviation** *(xsymbols)*

\[
\text{lesubprod} :: \ 'a \times \ 'b \Rightarrow (\ 'a \Rightarrow \ 'a \Rightarrow \ bool) \Rightarrow (\ 'b \Rightarrow \ 'b \Rightarrow \ bool) \Rightarrow \ 'a \times \ 'b \Rightarrow \ bool
\]

\[
((- \mathbb{N} (- \cdot) -) \ [50, \ 0, \ 0, \ 51] \ 50) \ where
\]

\[
p \subseteq (r_A, r_B) \ q == p \subseteq \text{Product.le} \ r_A r_B \ q
\]

**lemma** *unfold-lesub-prod: x \subseteq (r_A, r_B) y = le r_A r_B x y*

**lemma** *le-prod-Pair-conv [iff]: ((a_1, b_1) \subseteq (r_A, r_B)) = (a_1 \subseteq r_A a_2 \& b_1 \subseteq r_B b_2)*

**lemma** *less-prod-Pair-conv:*

\[
((a_1, b_1) \subseteq \text{Product.le} \ r_A r_B (a_2, b_2)) = (a_1 \subseteq r_A a_2 \& b_1 \subseteq r_B b_2)
\]

**lemma** *order-le-prod [iff]: order(\text{Product.le} \ r_A r_B) = (order \ r_A \& order \ r_B)*

**lemma** *acc-le-prodI [intro!]:*

\[
\{ \text{acc} \ r_A; \text{acc} \ r_B \} \Longrightarrow \text{acc(\text{Product.le} \ r_A r_B)}
\]

**lemma** *closed-lift2-sup:*

\[
\{ \text{closed} \ (\text{err} \ A); \text{closed} \ (\text{err} \ B) \ (\text{lift2} \ f) \} \Longrightarrow \\
\{ \text{closed} \ (\text{err}(A \times B)); \text{lift2}(\text{sup} \ f \ g) \}
\]

**lemma** *unfold-plus-sub-lift2: e_1 \cup_{\text{lift2}} e_2 = \text{lift2} f e_1 e_2*

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

\[
\text{assumes} \ x \in A \ y \in A \ \text{semilat}(\text{err} \ A, \ Err.le \ r, \ \text{lift2} \ f) \\
\text{shows} \ (x \cup y = \text{Err}) = (\neg (\exists z \in A. \ x \subseteq r z \& y \subseteq r z))
\]

**lemma** *err-semilat-Product-esl:*

\[
\bigwedge L_1 L_2. \ [ \text{err-semilat} \ L_1; \text{err-semilat} \ L_2 ] \Longrightarrow \text{err-semilat(\text{Product.esl} \ L_1 L_2)}
\]

**end**

### 4.5 Fixed Length Lists

**theory** *Listn*

**imports** *Err*

**begin**

**definition** *list :: nat \Rightarrow \ 'a \ set \Rightarrow \ 'a \ list \ set*

**where**

\[
\text{list} \ n \ A = \{ \text{xs. \ size} \ \text{xs} = n \& \text{set} \ \text{xs} \subseteq A \}
\]

**definition** *le :: \ 'a \ ord \Rightarrow \ ('a \ list) \ ord*

**where**

\[
\text{le} \ r = \text{list-all2} \ (\lambda x y. \ x \subseteq r y)
\]

**abbreviation** *(xsymbols)*

\[
\text{lesublist} :: \ 'a \ list \Rightarrow \ 'a \ ord \Rightarrow \ 'a \ list \Rightarrow \ bool \ ((- \mathbb{N} (- \cdot) -) \ [50, \ 0, \ 0, \ 51] \ 50) \ where
\]

\[
x \subseteq r y == x <=c (\text{Listn.le} \ r) y
\]

**abbreviation** *(xsymbols)*

\[
\text{lessublist} :: \ 'a \ list \Rightarrow \ 'a \ ord \Rightarrow \ 'a \ list \Rightarrow \ bool \ ((- \mathbb{N} (- \cdot) -) \ [50, \ 0, \ 0, \ 51] \ 50) \ where
\]

\[
x \subseteq r y == x <c (\text{Listn.le} \ r) y
\]

**definition** *map2 :: (\ 'a \Rightarrow \ 'b \Rightarrow \ 'c) \Rightarrow \ 'a \ list \Rightarrow \ 'b \ list \Rightarrow \ 'c \ list*
Theory Listn

where
map2 f = (λxs ys. map (split f) (zip xs ys))

abbreviation (xsymbols)
plussublist :: 'a list ⇒ ('a ⇒ 'b ⇒ 'c) ⇒ 'b list ⇒ 'c list
((·/½[⅓])·) [65, 0, 66] 65) where
x ⊔ f y == x ⊔ map2 f y

primrec coalesce :: 'a err list ⇒ 'a list err
where
coalescope [] = 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 = (λxs ys. if size xs = size ys then coalesce(xs ⊔ f ys) else Err)

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

lemmas [simp] = set-update-subset1

lemma unfold-lesub-list: xs ⊔ r ys = Listn.le r xs ys
lemma Nil-le-conv [iff]: ([]) ⊔ r ys = (ys = [])
lemma Cons-nolte-Nil [iff]: ¬ x#xs ⊔ r []
lemma Cons-le-Cons [iff]: x#xs ⊔ r y#ys = (x ⊔ r y ∧ xs ⊔ r ys)
lemma Cons-less-Cons [simp]:
| order r ⇒ x#xs ⊔ r y#ys = (x ⊔ r y ∧ xs ⊔ r ys) ∨ x = y ∧ xs ⊔ r ys
lemma list-update-le-cong:
| 1<size xs; xs ⊔ r ys; x ⊔ r y ⇒ xs[i:=x] ⊔ r ys[i:=y]

lemma le-listD: [ xs ⊔ r ys; p < size xs ] ⇒ xs!p ⊔ r ys!p
lemma le-list-refl: ∀ x. x ⊔ r x ⇒ xs ⊔ r xs
lemma le-list-trans: [ order r; xs ⊔ r ys; ys ⊔ r zs ] ⇒ xs ⊔ r zs
lemma le-list-antisym: [ order r; xs ⊔ r ys; ys ⊔ r zs ] ⇒ xs = ys
lemma order-listI [simp, intro!]: order r ⇒ order(Listn.le r)
lemma lesub-list-impl-same-size [simp]: xs ⊔ r ys ⇒ size ys = size xs
lemma lesssub-lengthD: xs ⊔ r ys ⇒ size ys = size xs
lemma le-list-appendI: a ⊔ r b ⇒ c ⊔ r d ⇒ a@&c ⊔ r b@d
lemma le-listI:
| assumes length a = length b
| assumes ∃a. n < length a ⇒ a!n ⊔ r b!n
| shows a ⊔ r b
lemma listI: [ size xs = n; set xs ⊆ A ] ⇒ xs ∈ list n A

lemma listE-length [simp]: xs ∈ list n A ⇒ size xs = n
lemma less-lengthI: [ xs ∈ list n A; p < n ] ⇒ p < size xs
lemma listE-set [simp]: \(xs \in \text{list } n \ A \implies \text{set } xs \subseteq A\)
lemma list-0 [simp]: \(\text{list } 0 \ A = \{\}\)
lemma in-list-Suc-iff:
  \((xs \in \text{list } (\text{Suc } n) \ A) = (\exists y \in A. \exists ys \in \text{list } n \ A. xs = y\#ys)\)
lemma Cons-in-list-Suc [iff]:
  \((x\#xs \in \text{list } (\text{Suc } n) \ A) = (x \in A \land xs \in \text{list } n \ A)\)
lemma list-not-empty:
  \(\exists a. a \in A \implies \exists xs. xs \in \text{list } n \ A\)
lemma nth-in [rule-format, simp]:
  \(\forall i \ n. \ \text{size } xs = n \implies xs \subseteq A \implies i < n \implies (xs!i) \in A\)
lemma listE-nth-in: \([xs \in \text{list } n \ A; \ i < n] \implies xs!i \in A\)
lemma listn-appendE [elim!]:
  \(l\#xs \in \text{list } n \ A \implies (\forall n'. n = \text{Suc } n' \implies l \in A \implies xs \in \text{list } n' \ A \implies P) \implies P\)
lemma listn-appendE [elim!]:
  \(a \@ b \in \text{list } n \ A \implies (\forall n \ n1. n = n1 + n2 \implies a \in \text{list } n1 \ A \implies b \in \text{list } n2 \ A \implies P) \implies P\)
lemma listt-update-in-list [simp, intro?]:
  \([xs \in \text{list } n \ A; x \in A] \implies xs[i := x] \in \text{list } n \ A\)
lemma list-appendI [intro?]:
  \([a \in \text{list } n \ A; b \in \text{list } m \ A] \implies a \@ b \in \text{list } (n+m) \ A\)
lemma list-map [simp]: \((\text{map } f \ xs \in \text{list } (\text{size } xs) \ A) = (f' \text{ set } xs \subseteq A)\)
lemma list-accumulateI [intro]: \(x \in A \implies \text{replicate } n \ x \in \text{list } n \ A\)
lemma plus-list-Cons [simp]:
  \((x\#xs) \ [\ y] \ ys = (\text{case } ys \ of \ [] \Rightarrow [] \ | \ y\#ys \Rightarrow (x \vartriangleright f \ ys)\#(\text{size } ys)\)\)
lemma length-plus-list [rule-format, simp]:
  \(\forall ys. \ \text{size}(xs \ [\ y] \ ys) = \min(\text{size } xs) \ (\text{size } ys)\)
lemma nth-plus-list [rule-format, simp]:
  \(\forall xs \ ys. \ i. \ \text{size } xs = n \implies \text{size } ys = n \implies i < n \implies (xs \ [\ y] \ ys)!i = (xs!i) \vartriangleright f \ (ys!i)\)
lemma (in Semilat) plus-list-ub1 [rule-format]:
  \([\text{set } xs \subseteq A; \text{set } ys \subseteq A; \text{size } xs = \text{size } ys]\)
  \(\implies xs \ [\ y] \ ys\)
lemma (in Semilat) plus-list-ub2:
  \([\text{set } xs \subseteq A; \text{set } ys \subseteq A; \text{size } xs = \text{size } ys]\)
  \(\implies ys \ [\ y] \ xs \ [\ y] \ ys\)
lemma (in Semilat) plus-list-lub [rule-format]:
  \(\forall xs \ ys. \ \text{set } xs \subseteq A \implies \text{set } ys \subseteq A \implies \text{set } zs \subseteq A\)
  \(\implies \text{size } zs = n \land \text{size } ys = n \implies \text{size } zs \ [\ y] \ ys \ [\ y] \ zs\)
lemma (in Semilat) list-update-incr [rule-format]:
  \(x \in A \implies \text{set } xs \subseteq A \implies \text{size } xs \subseteq x \implies (\forall i. \ i < \text{size } xs \implies \text{xs}[i := x \triangleright f \ \text{xs}!i])\)
lemma acc-le-listI [intro!]:
  \([\text{order } r; \text{acc } r]\) \implies \text{acc}(\text{Listn}.le \ r)\)
lemma closed-listI:
  \(\text{closed } S \ f \implies \text{closed } (\text{list } n \ S) \ (\text{map2 } f)\)
lemma Listn-sl-aux:
  \(\text{assumes } \text{Semilat } A \ r \ f \ \text{shows } \text{semilat } (\text{Listn}.sl \ n \ (A,r,f))\)
lemma Listn-sl: \(\text{semilat } L \implies \text{semilat } (\text{Listn}.sl \ n \ L)\)
lemma coalesce-in-err-list [rule-format]:
  \(\forall xes. \ xes \in \text{list } n \ (\text{err } A) \implies \text{coalesce } xes \in \text{err}(\text{list } n \ A)\)
lemmas lemma lem: \( \forall xs, x \sqcup_{op} \neq xs = x \# xs \)
lemmas lemma coalesce-eq-OK1-D [rule-format]:
\[
\text{semilat}(\text{err } A, \text{Err.le } r, \text{lif2 } f) \implies \\
\forall zs. \ xs \in \text{list } n \ A \implies (\forall ys. \ ys \in \text{list } n \ A \implies \\
(\forall zs. \ \text{coalesce } (xs [\sqcup]^f ys) = \text{OK } zs \implies zs [\sqcup] \ zs))
\]
lemmas lemma coalesce-eq-OK2-D [rule-format]:
\[
\text{semilat}(\text{err } A, \text{Err.le } r, \text{lif2 } f) \implies \\
\forall zs. \ xs \in \text{list } n \ A \implies (\forall ys. \ ys \in \text{list } n \ A \implies \\
(\forall zs. \ \text{coalesce } (xs [\sqcup]^f ys) = \text{OK } zs \implies ys [\sqcup] \ zs))
\]
lemmas lemma lif2-le-ub:
\[
[[ \text{semilat}(\text{err } A, \text{Err.le } r, \text{lif2 } f); \ x \in A; \ y \in A; \ x \sqcup_f y = \text{OK } z; \\
u \in A; \ z \sqsubseteq_r u \implies z \sqsubseteq_r u \]
\]
lemmas lemma coalesce-eq-ErrD [rule-format]:
\[
[[ \text{semilat}(\text{err } A, \text{Err.le } r, \text{lif2 } f); \ x \in A; \ y \in A ] \\
\implies (\exists u \in A. \ \ x \ sqsubseteq_r u \wedge y \ sqsubseteq_r u)
\]
lemmas lemma closed-err-lif2-conv:
\[
\text{closed } (\text{err } A) (\text{lif2 } f) = (\forall x \in A. \ \forall y \in A. \ x \sqcup_f y \in \text{err } A)
\]
lemmas lemma closed-map2-list [rule-format]:
\[
\text{closed } (\text{err } A) (\text{lif2 } f) \implies \\
\forall xs. \ xs \in \text{list } n \ A \implies (\forall ys. \ ys \in \text{list } n \ A \implies \\
\text{map2 } f \ ys \ xs \in \text{list } n \ (\text{err } A))
\]
lemmas lemma closed-lif2-sup:
\[
\text{closed } (\text{err } A) (\text{lif2 } f) \implies \\
\text{closed } (\text{err } \ (\text{list } n \ A)) (\text{lif2 } (\text{sup } f))
\]
lemmas lemma err-semilat-sup:
\[
\text{err-semilat } (A,r,f) \implies \\
\text{err-semilat } (\text{list } n \ A, \text{Lista.le } r, \text{sup } f)
\]
lemmas lemma err-semilat-upto-esl:
\[
\forall L. \ \text{err-semilat } L \implies \text{err-semilat } (\text{upto-esl } m \ L)
\]
end

4.6 Typing and Dataflow Analysis Framework

theory Typing-Framework imports Semilattices begin

  The relationship between dataflow analysis and a well-typed-instruction predicate.

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

definition stable :: 's ord \Rightarrow 's step-type \Rightarrow 's list \Rightarrow nat \Rightarrow bool
where
  stable r step τs p \isasymleftarrow (\forall (q,τ) \in \text{set } (\text{step } p (τ!p)).\ τ \sqsubseteq_r τ!q)
definition stables :: 's ord ⇒ 's step-type ⇒ 's list ⇒ bool
  where
  stables r step τ s ←→ (∀ p < size τ s. stable r step τ s p)

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

definition is-bcv :: 's ord ⇒ 's ⇒ 's step-type ⇒ nat ⇒ 's list ⇒ 's list ⇒ bool
  where
  is-bcv r T step n A bcv ←→ bounded step n A

4.7 More on Semilattices

theory SemilatAlg
imports Typing-Framework
begin

consts
  lesubstep-type :: (nat × 's) set ⇒ 's ord ⇒ (nat × 's) set ⇒ bool
notation (xsymbols)
  lesubstep-type ((− / {⊑} −) [50, 0, 51]) 50
  lesubstep-type-def:
  A {∈r} B ≡ ∀ (p,τ) ∈ A. ∃ τ′. (p,τ') ∈ B τ ⊑ r τ'

primrec pluslussub :: 'a list ⇒ ('a ⇒ 'a ⇒ 'a) ⇒ 'a ⇒ 'a
  where
  pluslussub [] f y = y
  | pluslussub (x#xs) f y = pluslussub xs f (x ∪ f y)
notation (xsymbols)
  pluslussub ((− / {∪} −) [65, 0, 66]) 50

definition bounded :: 's step-type ⇒ nat ⇒ bool
  where
  bounded step n ←→ (∀ p<n. ∀ τ. ∀ (q,τ') ∈ set (step p τ). q<n)

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

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

lemma [iff]: {} {∈r} B

lemma [iff]: (A {∈r} {}) = (A = {})

lemma lesubstep-union:
  [ A1 {∈r} B1; A2 {∈r} B2 ] ⇒ A1 ∪ A2 {∈r} B1 ∪ B2
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 ⇒ 's err step-type ⇒ 's err list ⇒ bool where
  wt-err-step r step τ s ≜ wt-step (Err.le r) Err step τ s

definition wt-app-eff :: 's ord ⇒ (nat ⇒ 's ⇒ bool) ⇒ 's step-type ⇒ 's list ⇒ bool where
  wt-app-eff r app step τ s ≜
  (∀ p < size τs. app p (τs!p)) ∧ (∀ (q,τ) ∈ set (step p (τs!p)). τ <= r τs!q))

definition map-snd :: ('b ⇒ 'c) ⇒ ('a × 'b) list ⇒ ('a × 'c) list
where
map-snd f = map (λ(x,y). (x, f y))

definition error :: nat ⇒ (nat × 'a err) list
where
error n = map (λx. (x, Err)) [0..< n]

definition err-step :: nat ⇒ (nat ⇒ 's ⇒ bool) ⇒ 's step-type ⇒ 's err step-type
where
err-step n app step p t =
(case t of
  Err ⇒ error n
| OK τ ⇒ if app p τ then map-snd OK (step p τ) else error n)

definition app-mono :: 's ord ⇒ (nat ⇒ 's ⇒ bool) ⇒ nat ⇒ 's set ⇒ bool
where
app-mono r app n A ←→ (∀s p t. s ∈ A ∧ p < n ∧ s ⊑ r t −→ app p t −→ 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) ∈ set (step p a) ] ⇒ q < n

lemma in-map-sndD: (a,b) ∈ set (map-snd f xs) ⇒ ∃ b′. (a,b′) ∈ set xs

lemma bounded-lift:
bounded step n ⇒ bounded (err-step n app step) n

lemma le-list-map-OK [simp]:
∀b. (map OK a [⊆Err,le r] map OK b) = (a [⊆r] b)

lemma map-snd-lessI:
set xs {⊆r} set ys ⇒ set (map-snd OK xs) {⊆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;
∀s p t. s ∈ A ∧ p < n ∧ s ⊆r t −→ app p t −→ set (step p s) {⊆r} set (step p t) ]
⇒ mono (Err.le r) (err-step n app step) n (err A)

lemma in-errorD: (x,y) ∈ set (error n) ⇒ y = Err

lemma pres-type-lift:
∀s ∈ A. ∀p. p < n −→ app p s −→ (∀q, s′ ∈ set (step p s). s′ ∈ A)
⇒ 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 \( wt\text{-app-eff} \) \( r \) \( app \) \( step \) \( (map \ \text{ok-value} \ ts) \)

**Lemma** \( wt\text{-app-eff-imp-wt-err} \):

- **Assumes** \( app\text{-eff} \): \( wt\text{-app-eff} \) \( r \) \( app \) \( step \) \( ts \)
- **Assumes bounded**: bounded \( (\text{err-step} \ (\text{size} \ ts) \ app \ step) \ (\text{size} \ ts) \)
- **Shows** \( wt\text{-err-step} \ r \ (\text{err-step} \ (\text{size} \ ts) \ app \ step) \ (\text{map} \ \text{OK} \ ts) \)

### 4.9 Kildall’s Algorithm

**Theory** Kildall

**Imports** SemilatAlg

**Begin**

**Primrec** propa :: \( 's \) binop \( \Rightarrow \) \( (\text{nat} \times 's) \) list \( \Rightarrow \) \( 's \) list \( \Rightarrow \) nat set \( \Rightarrow \) \( 's \) list \( \times \) nat set

**Where**

\[
\text{propa } f \ [] \ \tau s \ \omega = (\tau s, \omega) \\
\mid \text{propa } f \ (q' \# q s) \ \tau s \ \omega = (\text{let } (q, \tau) = q'; \ u = \tau \sqcup f \ \tau s q; \ w' = (\text{if } u = \tau s q \text{ then } w \text{ else insert } q w) \text{ in } \text{propa } f \ q s (\tau s[q := u]) \ w')
\]

**Definition** iter :: \( 's \) binop \( \Rightarrow \) \( 's \) step-type \( \Rightarrow \) \( 's \) list \( \Rightarrow \) nat set \( \Rightarrow \) \( 's \) list \( \times \) nat set

**Where**

\[
\text{iter } f \ \text{step } \tau s \ \omega = \text{while } (\lambda(\tau s, \omega). \ \omega \neq \{\}) \ (\lambda(\tau s, \omega). \ \text{let } p = \text{SOME } p. \ p \in \omega \text{ in } \text{propa } f \ \text{step } (\tau s p)) \ \tau s \ (\omega - \{p\})
\]

**Definition** unstables :: \( 's \) ord \( \Rightarrow \) \( 's \) step-type \( \Rightarrow \) \( 's \) list \( \Rightarrow \) nat set

**Where**

\[
\text{unstables } r \ \text{step } \tau s = \{p. \ p < \text{size } \tau s \wedge \neg \text{stable } r \ \text{step } \tau s \ p\}
\]

**Definition** kildall :: \( 's \) ord \( \Rightarrow \) \( 's \) binop \( \Rightarrow \) \( 's \) step-type \( \Rightarrow \) \( 's \) list \( \Rightarrow \) \( 's \) list

**Where**

\[
\text{kildall } r \ f \ \text{step } \tau s = \text{fst}(\text{iter } f \ \text{step } \tau s \ (\text{unstables } r \ \text{step } \tau s))
\]

**Primrec** merges :: \( 's \) binop \( \Rightarrow \) \( (\text{nat} \times 's) \) list \( \Rightarrow \) \( 's \) list \( \Rightarrow \) \( 's \) list

**Where**

\[
\text{merges } f \ [] \ \tau s = \tau s \\
\mid \text{merges } f \ (p' \# p s) \ \tau s = (\text{let } (p, \tau) = p' \text{ in } \text{merges } f \ p s (\tau s[p := \tau \sqcup f \ \tau s p]))
\]

**Lemmas** \( \text{[simp]} = \text{Let-def Semilat.le-iff-plus-unchanged} \ [\text{OF Semilat.intro, symmetric}] \)

**Lemma** \( \text{in Semilat}\) nth-merges:

\[
\forall ss. \ p < \text{length } ss; \ ss \in \text{list } n A; \forall (p, t) \in \text{set } ps. \ p < n \wedge t \in A \implies (\text{merges } f \ ps \ ss)!p = \text{map snd } (\{(p', t') \leftarrow ps. \ p' = p\} \cup f \ ss!p)
\]
(is \(\forall ss. [\cdot]; \cdot ?steptype ps] \implies ?P ss ps)\)

**Lemma** length-merges [simp]:
\[\forall ss. \text{size}(\text{merges } f \text{ ps } ss) = \text{size } ss\]

**Lemma** (in Semilat) merges-preserves-type-lemma:
**Shows** \(\forall xs. xs \in \text{list } n A \implies (\forall (p,x) \in \text{set } ps. \ p < n \wedge x \in A) \implies \text{merges } f \text{ ps } xs \in \text{list } n A\)

**Lemma** (in Semilat) merges-preserves-type [simp]:
\[\forall (p,x) \in \text{set } ps. \ p < n \wedge x \in A \implies \text{merges } f \text{ ps } xs \in \text{list } n A\]

**By** (simp add: merges-preserves-type-lemma)

**Lemma** (in Semilat) merges-incr-lemma:
\[\forall xs. xs \in \text{list } n A \implies (\forall (p,x) \in \text{set } ps. \ p < \text{size } xs \wedge x \in A) \implies \text{merges } f \text{ ps } xs\]

**Lemma** (in Semilat) merges-incr:
\[\forall (p,x) \in \text{set } ps. \ p < \text{size } xs \wedge x \in A \implies \text{merges } f \text{ ps } xs\]

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

**Lemma** (in Semilat) merges-same-conv [rule-format]:
\[(\forall xs. xs \in \text{list } n A \implies (\forall (p,x) \in \text{set } ps. \ p < \text{size } xs \wedge x \in A) \implies (\text{merges } f \text{ ps } xs = xs) = (\forall (p,x) \in \text{set } ps. \ x \in r \text{ xs}[p])\]

**Lemma** (in Semilat) list-update-le-listI [rule-format]:
\[\text{set } xs \subseteq A \implies \text{set } ys \subseteq A \implies \text{xs } \subseteq r \text{ ys}[p] \implies p < \text{size } xs \implies x \in r \text{ ys}[p] \implies x \in \text{set } (\text{size } ys \text{ ss}) \subseteq r \text{ ss}[p] \text{ ss}[p] \subseteq A \wedge p < \text{size } ts \text{ ss } \subseteq r \text{ ss}[p] \text{ ts}\]

**Lemma** (in Semilat) merges-pres-le-ab:
**Assumes** \(\text{set } ts \subseteq A \) \(\text{set } ss \subseteq A\)
\[\forall (p,t) \in \text{set } ps. \ t \subseteq r \text{ ts}[p] \wedge t \in A \wedge p < \text{size } ts \text{ ss } \subseteq r \text{ ss}[p] \text{ ts}\]

**Shows** \(\text{merges } f \text{ ps } ss \subseteq r \text{ ts}\)

**Lemma** decomp-propa:
\[\forall ss \ w. \ (\forall (q,t) \in \text{set } qs. \ q < \text{size } ss) \implies \text{propa } f \text{ qs } ss \ w = (\text{merges } f \text{ ss } \{q, \exists t. (q,t) \in \text{set } qs \wedge t \equiv f \text{ ss}[q] \equiv f \text{ ss}[q] \} \cup w)\]

**Lemma** (in Semilat) stable-pres-lemma:
**Shows** \[\text{pres-type } n A; \ \text{bounded } step n;\]
\[\text{ss } \in \text{list } n A; \ p \in w; \ \forall q \in w. \ q < n;\]
\[\forall q. \ q < n \implies q \not\in w \implies \text{stable } r \text{ step } ss \ q; \ q < n;\]
\[\forall s', (q,s') \in \text{set } \text{(step } p (\text{ss}[p]) \implies s' \equiv f \text{ ss}[q] = ss[q];\]
\[q \not\in w \ \forall q = p \]\n\[\implies \text{stable } r \text{ step } (\text{merges } f \text{ (step } p (\text{ss}[p]) \text{ ss}) \ q)\]

**Lemma** (in Semilat) merges-bounded-lemma:
[\(\text{mono } r \text{ step } n A; \text{ bounded } step n;\)
\[\forall (p',s') \in \text{set } \text{(step } p (\text{ss}[p]) \text{ s'}) \in A; \text{ ss } \in \text{list } n A; \text{ ts } \in \text{list } n A; \ p < n; \text{ ss } \subseteq r \text{ ts}; \forall p. \ p < n \implies \text{stable } r \text{ step } ts \ p\]
4.10 The Lightweight Bytecode Verifier
if $s \subseteq r$ 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 step-type ⇒ nat ⇒ 's ⇒ 's

where
  wtl-inst-list [] cert f r T B step pc s = s
  | wtl-inst-list (i#is) cert f r T B step pc s =
    (let $s' = wtl-cert cert f r T B step pc s in
      if $s' = T ∨ s = T then T else wtl-inst-list is cert f r T B step
      (pc + 1) $s' )

definition cert-ok :: 's certificate ⇒ nat ⇒ 's ⇒ 's ⇒ 's set ⇒ bool

where
  cert-ok cert n T B A ←→ (∀ i < n. cert!i ∈ A ∧ cert!i ≠ T) ∧ (cert!n = B)

definition bottom :: 'a ord ⇒ 'a ⇒ bool

where
  bottom r B ←→ (∀ x. B ⊑ r x)

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

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

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

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

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

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

lemma (in lbv) wtc:
  wtc c pc s = (if c!pc = ⊥ then wti c pc s else if s ⊆ r c!pc then wti c pc (c!pc) else ⊤)

lemma cert-okD1 [intro?]:
  cert-ok c n T B A ⊢ pc < n ⊢ c!pc ∈ A

lemma cert-okD2 [intro?]:
  cert-ok c n T B A ⊢ c!n = B

lemma cert-okD3 [intro?]:
cert-ok c n T B A \implies B \in A \implies pc < n \implies \text{c!Suc}\; pc \in A

\text{lemma cert-okD4\ [intro]}:\quad \text{cert-ok c n T B A} \implies pc < n \implies \text{c!pc} \neq T

\text{declare Let-def [simp]}

4.10.1 more semilattice lemmas

\text{lemma (in lbv) sup-top [simp, elim]}:\quad \text{assumes} x: x \in A \\
\text{shows} \; x \cup_f \top \; = \; \top

\text{lemma (in lbv) plusplusup-top [simp, elim]}:\quad \text{set} \; xs \subseteq A \implies xs \cup_f \top \; = \; \top \\
\text{by (induct xs) auto}

\text{lemma (in Semilat) pp-ub1:\quad}\text{assumes} \; x: x \in A \\
\text{shows} \; x \sqcup f \top \; = \; \top

\text{lemma (in lbv) le-bottom [simp, intro]}:\quad \bot \sqsubseteq_r x \\
\text{by (blast intro: antisym-r)}

4.10.2 merge

\text{lemma (in lbv) merge-Nil [simp]}:\quad \text{merge} \; c \; pc \; [] \; x \; = \; x \\
\text{by (simp add: mrg-def)}

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

\text{lemma (in lbv) merge-Err [simp]}:\quad \text{snd'set} \; ss \subseteq A \implies \text{merge} \; c \; pc \; ss \; x \; = \; \top \\
\text{by (induct ss) auto}

\text{lemma (in lbv) merge-not-top}:
\quad \forall x. \; \text{snd'set} \; ss \subseteq A \implies \text{merge} \; c \; pc \; ss \; x \neq \top \implies \\
\quad \forall (pc',s') \in \text{set} \; ss. \; (pc' \neq pc + 1 \implies \text{snd} \; s' \sqsubseteq_r c!pc' ) \\
\quad \text{by (simp add: mrg-def split-beta)}

\text{lemma (in lbv) merge-def}\;:\quad \text{shows} \\
\quad \forall x. \; x \in A \implies \text{snd'set} \; ss \subseteq A \implies \\
\quad \text{merge} \; c \; pc \; ss \; x = \\
\quad (\text{if} \; \forall (pc',s') \in \text{set} \; ss. \; pc' \neq pc + 1 \implies s' \sqsubseteq_r c!pc' \; \text{then} \\
\quad \quad \text{map} \; \text{snd} \; [(p',t') \leftarrow \text{ss}. \; p' = pc + 1] \cup_f x \\
\quad \quad \text{else} \; \top)
lemma (in lbv) merge-not-top-s:
  assumes x: \(x \in A\) and ss: snd'set ss \(\subseteq A\)
  assumes m: merge c pc ss x \(\neq \top\)
  shows merge c pc ss x = (map snd [(p', t') \leftarrow ss. p' = \text{pc} + 1]) \bigcup f x

4.10.3 wtl-inst-list

lemmas [iff] = not-Err-eq

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

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

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

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

lemma (in lbv) wtl-not-top:
  wtl ls c pc s \(\neq \top\)
  (cases s = \top) auto

lemma (in lbv) wtl-append [simp]:
  \(\forall\) 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 \(\neq \top\)
  (is \(?\) ? wtl is \(\neq - \implies -\))
lemma take-Suc:
  \(\forall\) n. n < length l \implies take (Suc n) l = (take n l)@(![l l] [is \(?\) P l])

lemma (in lbv) wtl-Suc:
  assumes suc: pc + 1 < length is
  assumes wtl: wtl (take pc is) c 0 s \(\neq \top\)
  shows wtl (take (pc + 1) is) c 0 s = wtc c pc (wtl (take pc is) c 0 s)

lemma (in lbv) wtl-all:
  assumes all: wtl is c 0 s \(\neq \top\)
  (is \(?\) wtl is \(\neq -\))
  assumes pc: pc < length is
  shows wtc c pc (wtl (take pc is) c 0 s) \(\neq \top\)

4.10.4 preserves-type

lemma (in lbv) merge-pres:
  assumes s0: snd'set ss \(\subseteq A\) and x: x \(\in A\)
  shows merge c pc ss x \(\in A\)
lemma pres-typeD2:
Theory LBVCorrect

pres-type step n A ⇒ s ∈ A ⇒ p < n ⇒ snd'set (step p s) ⊆ A
by auto (drule pres-typeD)

lemma (in lbv) wti-pres [intro?]::
assumes pres: pres-type step n A
assumes cert: c!(pc+1) ∈ A
assumes s-pc: s ∈ A pc < n
shows wti c pc s ∈ A

lemma (in lbv) wtc-pres::
assumes pres-type step n A
assumes c!pc ∈ A and c!(pc+1) ∈ A
assumes s ∈ A and pc < n
shows wtc c pc s ∈ A

lemma (in lbv) wtl-pres::
assumes pres-type step (length is) A
assumes cert-ok c (length is) ⊤⊥ A
assumes s: s ∈ A
assumes all: wtl is c 0 s ≠ ⊤
shows pc < length is ⇒ wtl (take pc ins) c 0 s ∈ A
(is ?len pc ⇒ ?wtl pc ∈ A)

end

4.11 Correctness of the LBV

theory LBVCorrect
imports LBVSpec Typing-Framework
begin

locale lbvs = lbv +
fixes s₀ :: 'a
fixes c :: 'a list
fixes ins :: 'b list
fixes τs :: 'a list
defines phi-def:
τs ≡ map (λpc. if c!pc = ⊥ then wtl (take pc ins) c 0 s₀ else c!pc)
[0...<size ins]

assumes bounded: bounded step (size ins)
assumes cert: cert-ok c (size ins) ⊤ ⊥ A
assumes pres: pres-type step (size ins) A

lemma (in lbvs) phi-None [intro?]::
[ pc < size ins; c!pc = ⊥ ] ⇒ τs!pc = wtl (take pc ins) c 0 s₀
lemma (in lbvs) phi-Some [intro?]::
[ pc < size ins; c!pc ≠ ⊥ ] ⇒ τs!pc = c!pc
lemma (in lbvs) phi-len [simp]: size τs = size ins
lemma (in lbvs) wtl-suc-pc::
assumes all: wtl ins c 0 s₀ ≠ ⊤
assumes pc: pc+1 < size ins
shows wtl (take (pc+1) ins) c 0 s₀ ⊑ r τs!(pc+1)
lemma (in lbvs) wtl-stable::
assumes wtl: wtl ins c 0 s₀ ≠ ⊤
assumes s₀: s₀ ∈ A and pc: pc < size ins
shows stable r step τ s pc
lemma (in lbvs) phi-not-top:
  assumes wtl: wtl ins c 0 s0 ≠ ⊤ and pc: pc < size ins
  shows τ s pc ≠ ⊤
lemma (in lbvs) phi-in-A:
  assumes wtl: wtl ins c 0 s0 ≠ ⊤ and s0 ∈ A
  shows τ s ∈ list (size ins) A
lemma (in lbvs) phi0:
  assumes wtl: wtl ins c 0 s0 ≠ ⊤ and s0 ∈ A
  shows ∀ pc < size ins. τ s pc ∈ list (size ins) A
theorem (in lbvs) wtl-sound:
  assumes wtl: wtl ins c 0 s0 ≠ ⊤ and s0 ∈ A
  assumes mono: mono r step (size τ s) A
  assumes pres: pres-type step (size τ s) A
  assumes τ s: ∀ pc < size τ s. τ s pc ∈ A ∧ τ s pc ≠ ⊤
  shows ∃ τ s ∈ list (size ins) A. wt-step r ⊤ step τ s ∧ s0 ⊑ τ 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 τ s pc' ←→ (∃ pc s'. pc' ≠ pc+1 ∧ pc < size τ s ∧ (pc',s') ∈ set (step pc (τ s pc')))
definition make-cert :: 's step-type ⇒ 's list ⇒ 's ⇒ 's certificate where
  make-cert step τ s B = map (λpc. if is-target step τ s pc then τ s! pc else B) [0..<size τ s] @ [B]
lemma [code]:
  is-target step τ s pc' =
  list-ex (λpc. pc' ≠ pc+1 ∧ List.member (map fst (step pc (τ s pc')))) pc' [0..<size τ s]
locale lbvc = lbv +
  fixes τ s :: 'a list
  fixes c :: 'a list
  defines cert-def: c ≡ make-cert step τ s ⊥

  assumes mono: mono r step (size τ s) A
  assumes pres: pres-type step (size τ s) A
  assumes τ s: ∀ pc < size τ s. τ s pc ∈ A ∧ τ s pc ≠ ⊤
  assumes bounded: bounded step (size τ s)
  assumes B-neq-T: ⊥ ≠ ⊤
lemma (in lbvc) cert: cert-ok c (size τ s) ⊥ ⊥ A
lemmas [simp del] = split-paired-Ex
lemma (in lbvc) cert-target [intro?]:
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:\n\((C,D) \in \text{subcls1}\ P \Longrightarrow \text{super}\ P\ C = D\)
by (unfold super-def) (auto dest: subcls1D)

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

definition sup :: 'c prog \Rightarrow\ ty \Rightarrow\ ty \Rightarrow\ ty\ err
where
\(\text{sup} P\ T_1\ T_2 \equiv\)
if is-refT \(T_1\) \& is-refT \(T_2\) then
OK (if \(T_1 = \text{NT}\) then \(T_2\) else
if \(T_2 = \text{NT}\) then \(T_1\) else
(Class (exec-lub (subcls1 P) (super P) (the-Class \(T_1\)) (the-Class \(T_2\))))
else
(if \(T_1 = T_2\) then OK \(T_1\) else Err)
by (simp add: sup-def fun-eq-iff)

abbreviation
subtype :: 'c prog \Rightarrow\ ty \Rightarrow\ ty \Rightarrow\ bool
where subtype \(P \equiv\) widen \(P\)

definition esl :: 'c prog \Rightarrow\ ty\ esl
where
\(\text{esl} P \equiv (\text{types} P, \text{subtype} P, \text{sup} P)\)

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

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

lemma widen-antisym:
\([\text{wf-prog}\ m\ P; P \vdash T \leq U; P \vdash U \leq T] \Longrightarrow T = U\)

lemma order-widen [intro,simp]:
\(\text{wf-prog}\ m\ P \Longrightarrow \text{order}\ (\text{subtype} P)\)

lemma NT-widen:
Theory JVM-SemiType

\[ P \vdash NT \leq T = (T = NT \vee (\exists C. T = Class C)) \]

**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:
\[ \text{wf } ((\text{subcls1 } P)^{-1}) \Rightarrow \text{acc } (\text{subtype } P) \]

**Lemma** wf-subtype-acc [intro, simp]:
\[ \text{wf-prog } wf\text{-mb } P \Rightarrow \text{acc } (\text{subtype } P) \]

**Lemma** exec-lub-refl [simp]:
\[ \text{exec-lub } r f T T = T \]

**Lemma** closed-err-types:
\[ \text{wf-prog } wf\text{-mb } P \Rightarrow \text{closed } (\text{err } (\text{types } P)) (\text{lift2 } (\text{sup } P)) \]

**Lemma** sup-subtype-greater:
\[
\begin{array}{l}
\text{[ wf-prog } \text{wf-mb } P; \text{is-type } P t1; \text{is-type } P t2; \text{sup } P t1 t2 = \text{OK } s ] \\
\Rightarrow \text{subtype } P t1 s \wedge \text{subtype } P t2 s
\end{array}
\]

**Lemma** sup-subtype-smallest:
\[
\begin{array}{l}
\text{[ wf-prog } \text{wf-mb } P; \text{is-type } P a; \text{is-type } P b; \text{is-type } P c; \\
\text{subtype } P a c; \text{subtype } P b c; \text{sup } P a b = \text{OK } d ] \\
\Rightarrow \text{subtype } P d c
\end{array}
\]

**Lemma** sup-exists:
\[
\begin{array}{l}
\text{[ subtype } P a c; \text{subtype } P b c ] \Rightarrow \text{EX } T. \text{sup } P a b = \text{OK } T
\end{array}
\]

**Lemma** err-semilat-JType-esl:
\[ \text{wf-prog } \text{wf-mb } P \Rightarrow \text{err-semilat } (\text{esl } P) \]

end

### 4.14 The JVM Type System as Semilattice

theory JVM-SemiType imports SemiType begin

type-synonym \( ty_t = ty \text{ err list} \)
type-synonym \( ty_s = ty \text{ list} \)
type-synonym \( ty_t = ty_t \times ty_t \)
type-synonym \( ty_i = ty_t \text{ option} \)
type-synonym \( ty_m = ty_i \text{ list} \)
type-synonym \( ty_P = \text{mname} \Rightarrow \text{cname} \Rightarrow ty_m \)

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

definition loc-sl :: 'c prog \Rightarrow nat \Rightarrow ty_t \text{ sl}
where
\[ \text{loc-sl } P \text{ mzl } \equiv \text{Listn.sl } mzl (\text{Err.sl } (\text{SemiType.esl } P)) \]

definition sl :: 'c prog \Rightarrow nat \Rightarrow nat \Rightarrow ty_t \text{ err sl}
where
\[ \text{sl } P \text{ mxs mzl } \equiv \\
\text{Err.sl(} \text{Opt.esl(Product.esl } (\text{stk-esl } P \text{ mxs}) \text{ (Err.esl(lc-sl } P \text{ mzl}))) \]

definition states :: 'c prog \Rightarrow nat \Rightarrow nat \Rightarrow ty_t \text{ err set}
where
\[ \text{states } P \text{ mxs mzl } \equiv \text{fst } (\text{sl } P \text{ mxs mzl}) \]
\[
\begin{align*}
\text{definition } le &:: \text{'}c prog \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow ty_i, \text{ err ord} \\
\text{where} & \\
le P \ mxs \ mxl &\equiv \text{fst}(\text{snd}(\text{sl } P \ mxs \ mxl))
\end{align*}
\]

\[
\begin{align*}
\text{definition } sup &:: \text{'}c prog \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow ty_i, \text{ err binop} \\
\text{where} & \\
sup P \ mxs \ mxl &\equiv \text{snd}(\text{snd}(\text{sl } P \ mxs \ mxl))
\end{align*}
\]

\[
\begin{align*}
\text{definition } sup-ty-opt &:: [\text{'}c prog, ty_i, ty_i] \Rightarrow \text{bool} \\
\text{where} & \\
sup-ty-opt P &\equiv \text{Err} \cdot le(\text{subtype } P)
\end{align*}
\]

\[
\begin{align*}
\text{definition } sup-state &:: [\text{'}c prog, ty_i, ty_i] \Rightarrow \text{bool} \\
\text{where} & \\
sup-state P &\equiv \text{Product} \cdot le(\text{Listn} \cdot le(\text{subtype } P))(\text{Listn} \cdot le(\text{sup-ty-opt } P))
\end{align*}
\]

\[
\begin{align*}
\text{definition } sup-state-opt &:: [\text{'}c prog, ty_i, ty_i] \Rightarrow \text{bool} \\
\text{where} & \\
sup-state-opt P &\equiv \text{Opt} \cdot le(\text{sup-state } P)
\end{align*}
\]

\[
\begin{align*}
\text{abbreviation } sup-loc &:: [\text{'}c prog, ty_i, ty_i] \Rightarrow \text{bool} \\
\text{where} & \\
\text{P } \vdash \text{ LT } \iff [\le ] \vdash T \iff [71,71,71] \ 70 &\equiv \text{list-all2} (\text{sup-ty-opt } P) \text{ LT } \text{ LT}'
\end{align*}
\]

\[
\begin{align*}
\text{lemma JVM-states-unfold}: & \\
\text{states } P \ mxs \ mxl &\equiv \text{err}(\text{opt}(\text{Union } \{\text{list } n \ (\text{types } P) \ | n. \ n \leq \text{ mxs}\}) <\ast>) \\
\text{list mxl } (\text{err}(\text{types } P)))
\end{align*}
\]

\[
\begin{align*}
\text{lemma JVM-le-unfold}: & \\
le P \ mxs \ mxl &\equiv \text{Err} \cdot le(\text{Opt} \cdot le(\text{Product} \cdot le(\text{Listn} \cdot le(\text{subtype } P)))(\text{Listn} \cdot le(\text{Err} \cdot le(\text{subtype } P)))))
\end{align*}
\]

\[
\begin{align*}
\text{lemma sl-def2}: & \\
JVM-SemiType.\text{sl } P \ mxs \ mxl &\equiv \ \\
(\text{states } P \ mxs \ mxl, \text{JVM-SemiType} \cdot \text{le } P \ mxs \ mxl, \text{JVM-SemiType} \cdot \text{sup } P \ mxs \ mxl)
\end{align*}
\]

\[
\begin{align*}
\text{lemma JVM-le-conv}: & \\
le P \ mx n \ (\text{OK } t1) \ (\text{OK } t2) &\equiv P \vdash \text{ t1 } \leq' \text{ t2}
\end{align*}
\]

\[
\begin{align*}
\text{lemma JVM-le-Err-conv}: & \\
le P \ mx n &\equiv \text{Err} \cdot le(\text{sup-state-opt } P)
\end{align*}
\]

\[
\begin{align*}
\text{lemma err-le-unfold } \iff &: \\
\text{Err} \cdot le \ (\text{OK } a) \ (\text{OK } b) &\equiv a \ b
\end{align*}
\]

4.14.1 Unfolding
4.14.2 Semilattice

**Lemma** order-sup-state-opt [intro, simp]:
\( P \implies \text{order (sup-state-opt P)} \)

**Lemma** semilat-JVM [intro]:
\( P \implies \text{semilat (JVM-SemiType.sl P mxs mzl)} \)

**Lemma** acc-JVM [intro]:
\( P \implies \text{acc (JVM-SemiType.le P mxs mzl)} \)

4.14.3 Widening with \( \top \)

**Lemma** subtype-refl [iff]: \( P \leq T \leq T \)

**Lemma** Err-any-conv [iff]: \( P \leq \text{Err} \leq T = (T = \text{Err}) \)

**Lemma** any-Err [iff]: \( P \leq T \leq \text{Err} \)

**Lemma** OK-OK-conv:
\( P \leq \text{OK} \leq \text{OK} \leq P \)

**Lemma** any-Err [iff]: \( P \leq \text{Err} \leq \text{Err} \)

**Lemma** sup-ty-opt-trans [intro?, trans]:
\( [\ P \leq a \leq c; P \leq b \leq c \implies P \leq a \leq c \)

4.14.4 Stack and Registers

**Lemma** stk-convert:
\( P \leq \text{ST} \leq \text{Listn} \leq \text{ST} \)

**Lemma** sup-loc-def [iff]: \( P \leq \text{LT} \leq \text{LT} \)

**Lemma** sup-loc-widens-conv [iff]:
\( P \leq \text{map} \text{OK} \text{Ts} \leq \text{map} \text{OK} \text{Ts'} = P \leq \text{Ts} \leq \text{Ts'} \)

**Lemma** sup-loc-trans [intro?, trans]:
\( [\ P \leq a \leq c; P \leq b \leq c \implies P \leq a \leq c \)

4.14.5 State Type

**Lemma** sup-state-conv [iff]:
\( P \leq (\text{ST},\text{LT}) \leq (\text{ST}',\text{LT}') = (P \leq \text{ST} \leq \text{ST}' \wedge P \leq \text{LT} \leq \text{LT}') \)

**Lemma** sup-state-conv2:
\( P \leq s1 \leq s2 = (P \leq \text{fst} s1 \leq \text{fst} s2 \wedge P \leq \text{snd} s1 \leq \text{snd} s2) \)

**Lemma** sup-state-refl [iff]: \( P \leq s \leq s \)

**Lemma** sup-state-trans [intro?, trans]:
\( [\ P \leq a \leq c; P \leq b \leq c \implies P \leq a \leq c \)

**Lemma** sup-state-opt-None-any [iff]:
\( P \leq \text{None} \leq s \)

**Lemma** sup-state-opt-any-None [iff]:
\( P \leq s \leq \text{None} = (s = \text{None}) \)

**Lemma** sup-state-opt-Some-Some [iff]:
\( P \leq \text{Some } a \leq \text{Some } b = P \leq a \leq b \)
lemma sup-state-opt-any-Some:
\[ P \vdash (\text{Some } s) \leq' X = (\exists s'. X = \text{Some } s' \land P \vdash s \leq_1 s') \]

lemma sup-state-opt-refl [iff]: \[ P \vdash s \leq s \]

lemma sup-state-opt-trans [intro?, trans]: \[ [P \vdash a \leq b; P \vdash b \leq' c] \implies P \vdash a \leq' c \]

4.15 Effect of Instructions on the State Type

theory Effect
imports JVM-SemiType ./JVM/JVMExceptions
begin
— FIXME
locale prog =
    fixes \( P :: 'a \)

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: \( mxl \equiv 1 + \text{size } Ts + mxl_0 \)

    Program counter of successor instructions:
primrec succs :: instr \Rightarrow ty_i \Rightarrow pc \Rightarrow pc list where
succe (Load idx) \tau pc = [pc+I]
| succs (Store idx) \tau pc = [pc+I]
| succs (Push v) \tau pc = [pc+I]
| succs (Getfield F C) \tau pc = [pc+I]
| succs (Putfield F C) \tau pc = [pc+I]
| succs (New C) \tau pc = [pc+I]
| succs (Checkcast C) \tau pc = [pc+I]
| succs Pop \tau pc = [pc+1]
| succs IAdd \tau pc = [pc+1]
| succs CmpEq \tau pc = [pc+1]
| succs-IfFalse: succs (IfFalse b) \tau pc = [pc+1, nat (int pc + b)]
| succs-Goto: succs (Goto b) \tau pc = [nat (int pc + b)]
| succs-Return: succs Return \tau pc = []
| succs-Invoke: succs (Invoke M n) \tau pc = (if (fst \tau)!n = NT then [] else [pc+I])
| succs-Throw: succs Throw \tau pc = []

Effect of instruction on the state type:
fun the-class:: ty \Rightarrow cname where
the-class (Class C) = C
fun eff_i :: instr × 'm prog × ty_i ⇒ ty_i where
  eff_i_Load:
  eff_i (Load n, P, (ST, LT)) = (ok-val (LT ! n) ≠ ST, LT)
| eff_i_Store:
  eff_i (Store n, P, (T#ST, LT)) = (ST, LT[n := OK T])
| eff_i_Push:
  eff_i (Push v, P, (ST, LT)) = (the (typeof v) ≠ ST, LT)
| eff_i_Getfield:
  eff_i (Getfield F C, P, (T#ST, LT)) = (snd (field P C F) ≠ ST, LT)
| eff_i_Putfield:
  eff_i (Putfield F C, P, (T_1#T_2#ST, LT)) = (ST,LT)
| eff_i_New:
  eff_i (New C, P, (ST,LT)) = (Class C ≠ ST, LT)
| eff_i_Checkcast:
  eff_i (Checkcast C, P, (T#ST,LT)) = (Class C ≠ ST,LT)
| eff_i_Pop:
  eff_i (Pop, P, (T#ST,LT)) = (ST,LT)
| eff_i_Add:
  eff_i (Add, P,(T_1#T_2#ST,LT)) = (Integer#ST,LT)
| eff_i_CmpEq:
  eff_i (CmpEq, P, (T_1#T_2#ST,LT)) = (Boolean#ST,LT)
| eff_i_IfFalse:
  eff_i (IfFalse b, P, (T_1#ST,LT)) = (ST,LT)
| eff_i_Invoke:
  eff_i (Invoke M n, P, (ST,LT)) =
  (let C = the-class (ST!n); (D,Ts,T_r,b) = method P C M
   in (T_r ≠ drop (n+1) ST, LT))
| eff_i_Goto:
  eff_i (Goto n, P, s) = s

fun is-relevant-class :: instr ⇒ 'm prog ⇒ cname ⇒ bool where
  rel_Getfield:
  is-relevant-class (Getfield F D) = (λP C. P ⊢ NullPointer ≤* C)
| rel_Putfield:
  is-relevant-class (Putfield F D) = (λP C. P ⊢ NullPointer ≤* C)
| rel_Checkcast:
  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 = e in is-relevant-class i P C ∧ pc ∈ {f..<t})

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

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

**definition norm-eff :: instr ⇒ 'm prog ⇒ nat ⇒ (pc \times ty_i)\text{ list where}**
\[ \text{norm-eff} i P pc \tau = \text{map } (\lambda p'. (p', \text{Some } (\text{eff } i P pc \tau))) \text{ (success } i \tau pc) \]

**definition eff :: instr ⇒ 'm prog ⇒ pc ⇒ ex-table ⇒ ty_i ⇒ (pc \times ty_i)\text{ list where}**
\[ \text{eff } i P pc et t = \text{case } t \text{ of} \]
\[ \text{None } \Rightarrow [] \]
\[ \text{Some } \tau \Rightarrow (\text{norm-eff } i P pc \tau) \&\& (\text{xcpt-eff } i P pc \tau et) \]

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

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

Conditions under which eff is applicable:

**fun appi :: instr \times 'm prog \times pc \times nat \times ty \times ty_i ⇒ bool where**
\[ \text{appi-Load:} \]
\[ \text{appi } (\text{Load } n, P, pc, mxs, T_r, (ST,LT)) = \]
\[ (n < \text{length } LT \land LT \land \text{length } ST < mxs) \]
\[ \text{| appi-Store:} \]
\[ \text{appi } (\text{Store } n, P, pc, mxs, T_r, (T \# ST, LT)) = \]
\[ (n < \text{length } LT) \]
\[ \text{| appi-Push:} \]
\[ \text{appi } (\text{Push } v, P, pc, mxs, T_r, (ST,LT)) = \]
\[ (\text{length } ST < mxs \land \text{typeof } v \neq \text{None}) \]
\[ \text{| appi-Getfield:} \]
\[ \text{appi } (\text{Getfield } F \ C, P, pc, mxs, T_r, (T \# ST, LT)) = \]
\[ (\exists T_f. P \vdash C \text{ sees } F: T_f \text{ in } C \land P \vdash T \leq \text{Class } C) \]
\[ \text{| appi-Putfield:} \]
\[ \text{appi } (\text{Putfield } F \ C, P, pc, mxs, T_r, (T_1 \# T_2 \# ST, LT)) = \]
\[ (\exists T_f. P \vdash C \text{ sees } F: T_f \text{ in } C \land P \vdash T_2 \leq (\text{Class } C) \land P \vdash T_1 \leq T_f) \]
\[ \text{| appi-New:} \]
\[ \text{appi } (\text{New } C, P, pc, mxs, T_r, (ST,LT)) = \]
\[ (\text{is-class } P \ C \land \text{length } ST < mxs) \]
\[ \text{| appi-Checkcast:} \]
\[ \text{appi } (\text{Checkcast } C, P, pc, mxs, T_r, (T \# ST,LT)) = \]
\[ (\text{is-class } P \ C \land \text{is-refT } T) \]
\[ \text{| appi-Pop:} \]
\[ \text{appi } (\text{Pop } P, pc, mxs, T_r, (T \# ST,LT)) = \]
\[ \text{true} \]
\[ \text{| appi-IAdd:} \]
\[ \text{appi } (\text{IAdd } P, pc, mxs, T_r, (T_1 \# T_2 \# ST,LT)) = (T_1 = T_2 \land T_1 = \text{Integer}) \]
\[ \text{| appi-CmpEq:} \]
\[ \text{appi } (\text{CmpEq } P, pc, mxs, T_r, (T_1 \# T_2 \# ST,LT)) = \]
\[ (T_1 = T_2 \lor \text{is-refT } T_1 \land \text{is-refT } T_2) \]
\[ \text{| appi-IfFalse:} \]
\[ \text{appi } (\text{IfFalse } b, P, pc, mxs, T_r, (\text{Boolean } \# ST,LT)) = \]
Theory Effect

\[(0 \leq \text{int } pc + b)\]

| \text{app}\_\text{-Goto}:
| \text{app}_i (\text{Goto } b, P, pc, mxs, T_r, s) =
| (0 \leq \text{int } pc + b)\
| \text{app}\_\text{-Return}:
| \text{app}_i (\text{Return } P, pc, mxs, T_r, (T \# ST, LT)) =
| (P \vdash T \leq T_r)\
| \text{app}\_\text{-Throw}:
| \text{app}_i (\text{Throw } P, pc, mxs, T_r, (T \# ST, LT)) =
| \text{is-ref } T T\
| \text{app}\_\text{-Invoke}:
| \text{app}_i (\text{Invoke } M n, P, pc, mxs, T_r, (ST, LT)) =
| (n < \text{length } ST \land
| (ST\!n \neq NT \rightarrow
| (\exists C D Ts T m. \text{Class } C \land P \vdash C \text{ sees } M:T s \rightarrow T = m \text{ in } D \land
| P \vdash \text{rev (take } n \text{ ST) } [\leq] \text{ Ts})))\
| \text{app}\_\text{-default}:
| \text{app}_i (i, P, pc, mxs, T_r, s) = False

definition \text{xcpt-app} :: \text{instr} \Rightarrow \text{'}m \text{ prog } \Rightarrow \text{pc } \Rightarrow \text{nat } \Rightarrow \text{ex-table } \Rightarrow \text{ty}_i \Rightarrow \text{bool } \text{where}
\text{xcpt-app } i \ P \ pc \ mxs \ xt \ \tau \longleftrightarrow (\forall \langle f, t, C, h, d \rangle \in \text{set (relevant-entries } P \ i \ pc \ xt \). \text{is-class } P \ C \land d \leq
\text{size (fst } t \rangle \land d < \text{mxs}))

definition \text{app} :: \text{instr} \Rightarrow \text{'}m \text{ prog } \Rightarrow \text{nat } \Rightarrow \text{ty } \Rightarrow \text{nat } \Rightarrow \text{nat } \Rightarrow \text{ex-table } \Rightarrow \text{ty}_i \Rightarrow \text{bool } \text{where}
\text{app } i \ P \ mxs \ T_r \ pc \ mpc \ xt \ \tau = (\text{case } t \text{ of None } \Rightarrow \text{True } | \text{Some } \tau \Rightarrow
\text{app}_i (i, P, pc, mxs, T_r, \tau) \land \text{xcpt-app } i \ P \ pc \ mxs \ xt \ \tau \land
(\forall \langle \text{pc}', \tau' \rangle \in \text{set (eff } i \ P \ pc \ xt \ \tau). \text{pc}' < \text{mpc}))

lemma \text{app-}:Some:
\text{app}_i (i, P, pc, mxs, T_r, \tau) =
(\text{app}_i (i, P, pc, mxs, T_r, \tau) \land \text{xcpt-app } i \ P \ pc \ mxs \ xt \ \tau \land
(\forall \langle \text{pc}', \tau' \rangle \in \text{set (eff } i \ P \ pc \ xt (\text{Some } \tau)). \text{pc}' < \text{mpc}))

by (simp add: app-def)

locale \text{eff} = jvm-method +
fixes \text{eff}_i and \text{app}_i and \text{eff} and \text{app}
fixes \text{norm-eff} and \text{xcpt-app} and \text{xcpt-eff}

fixes \text{mpc}
defines \text{mpc } \equiv \text{size is}

defines \text{eff}_i, i \ \tau \equiv \text{Effect.eff}_i (i, P, \tau)
notes \text{eff}_i-simps [simp] = \text{Effect.eff}_i-simps [where } P = P, \text{folded eff}_i-\text{def]}

defines \text{app}_i, i \ \text{pc} \ \tau \equiv \text{Effect.app}_i (i, P, pc, mxs, T_r, \tau)
notes \text{app}_i-simps [simp] = \text{Effect.app}_i-simps [where } P=P \text{ and } mxs=mxs \text{ and } T_r=T_r, \text{folded app}_i-\text{def]}

defines \text{xcpt-eff} i \ \text{pc} \ \tau \equiv \text{Effect.xcpt-eff} i \ P \ pc \ \tau \ xt
notes \text{xcpt-eff} = \text{Effect.xcpt-eff-def [of } - P - - xt, \text{folded xcpt-eff-def]}

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

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

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

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

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

lemma length-cases3:
assumes ⋀ LT. P ([],LT)
assumes ⋀ l LT. P ([l],LT)
assumes ⋀ l ST LT. P (l#ST,LT)
shows P s

lemma length-cases4:
assumes ⋀ LT. P ([],LT)
assumes ⋀ l LT. P ([l],LT)
assumes ⋀ l' LT. P ([l'],LT)
assumes ⋀ l' ST LT. P (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 (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 (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 (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 (Getfield F C,P,pc, mxs, T r, s) =
(∃ oT vT ST LT. s = (oT#ST, LT) ∧
Theory Effect

\[ P \vdash C \text{ sees } F : vT \text{ in } C \land P \vdash oT \leq (\text{Class } C) \]

by (rule length-cases2 [of - s]) auto

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

lemma appNew [simp]:
\[ \text{app}_i (\text{New } C P, pc, mxs, T_r, s) = (\exists ST LT. s = (ST, LT) \land \text{is-class } P C \land \text{length } ST < mxs) \]
by (cases s, simp)

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

lemma appIPop [simp]:
\[ \text{app}_i (\text{Pop } P, pc, mxs, T_r, s) = (\exists ts ST LT. s = (ts \# ST, LT)) \]
by (rule length-cases2, auto)

lemma appIAdd [simp]:
\[ \text{app}_i (\text{IAdd } P, pc, mxs, T_r, s) = (\exists ST LT. s = (\text{Integer } \# \text{Integer } \# ST, LT)) \]

lemma appIFalse [simp]:
\[ \text{app}_i (\text{IfFalse } b P, pc, mxs, T_r, s) = (\exists ST LT. s = (\text{Boolean } \# ST, LT) \land 0 \leq \text{int } pc + b) \]

lemma appICmpEq [simp]:
\[ \text{app}_i (\text{CmpEq } P, pc, mxs, T_r, s) = (\exists T_1 T_2 ST LT. s = (T_1 \# T_2 \# ST, LT) \land (\neg \text{is-refT } T_1 \land T_2 = T_1 \lor \text{is-refT } T_1 \land \text{is-refT } T_2)) \]
by (rule length-cases4, auto)

lemma appReturn [simp]:
\[ \text{app}_i (\text{Return } P, pc, mxs, T_r, s) = (\exists T ST LT. s = (T \# ST, LT) \land P \vdash T \leq T_r) \]
by (rule length-cases2, auto)

lemma appIThrow [simp]:
\[ \text{app}_i (\text{Throw } P, pc, mxs, T_r, s) = (\exists T ST LT. s = (T \# ST, LT) \land \text{is-refT } T) \]
by (rule length-cases2, auto)

lemma effNone:
\[ (pc', s') \in \text{set } (\text{eff } i P pc \text{ et None}) \implies s' = \text{None} \]
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]:
\[ \text{relevant-entries } P \text{ i } pc (xt @ xt') = \text{relevant-entries } P \text{ i } pc \text{ xt } @ \text{relevant-entries } P \text{ i } pc \text{ xt}' \]
by (unfold relevant-entries-def) simp

lemma xcpt-app-append [iff]:
\[ \text{xcpt-app } i P pc mxs (xt @ xt') \tau = (\text{xcpt-app } i P pc mxs xt \tau \land \text{xcpt-app } i P pc mxs xt' \tau) \]
by (unfold xcpt-app-def) fastforce
lemma xcpt-eff-append [simp]:
xctp-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 ⊢ τ ≤ᵣ τ'
shows appᵣ (i,P,mxs,mpc,rT,τ') → appᵣ (i,P,mxs,mpc,rT,τ)

lemma succs-mono:
assumes wf: wf-prog p P and appᵣ: appᵣ (i,P,mxs,mpc,rT,τ')
shows P ⊢ τ ≤ᵣ τ' → set (succs i τ pc) ⊆ set (succs i τ' pc)

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

lemma effᵣ-i-mono:
assumes wf: wf-prog p P
assumes less: P ⊢ τ ≤ᵣ τ'
assumes appᵣ: app i P m rT pc mpc xt (Some τ')
assumes succs: succs i τ pc ≠ [] succs i τ' pc ≠ []
shows P ⊢ effᵣ (i,P,τ) ≤ᵣ effᵣ (i,P,τ')

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 ⇒ tyᵣ i err list ⇒ bool
where
check-types P mxs mxl τs ≡ set τs ⊆ states P mxs mxl
— An instruction is wellytipped if it is applicable and its effect
— is compatible with the type at all successor instructions:

**definition**

\[ wt-instr :: \left[ 'm \text{ prog,ty,nat,pc,ex-table,instr,pc,ty}_m \right] \Rightarrow \text{bool} \]

\[ (\ldots, \ldots, \vdash \ldots :: - [60,0,0,0,0,0,0,61] 60) \]

**where**

\[ P,T,mxs,mpc,xt \vdash i,pc :: \tau_s \equiv \]

\[ \text{app } i \ P \ mxs \ T \ pc \ mpc \ xt \ (\tau_s!pc) \land \]

\[ (\forall (pc', \tau') \in \text{set } (\text{eff } i \ P \ pc \ xt \ (\tau_s!pc)). \ P \vdash \tau' \leq \tau_s!pc') \]

— The type at \( pc=0 \) conforms to the method calling convention:

**definition**

\[ wt-start :: \left[ \ '	ext{m} \text{ prog},\text{cname,ty list,nat,ty}_m \right] \Rightarrow \text{bool} \]

**where**

\[ wt-start P C Ts mxl 0 \tau_s \equiv \]

\[ \ P \vdash \text{Some } (\text{[]}, \text{OK } \text{(Class } C \text{)}# \text{map } \text{OK } \tau_s @\text{replicate } mxl_0 \text{ Err}) \leq \tau_s!0 \]

— A method is welltyped if the body is not empty,
  — if the method type covers all instructions and mentions
  — declared classes only, if the method calling convention is respected, and
  — if all instructions are welltyped.

**definition**

\[ wt-method :: \left[ \ '	ext{m} \text{ prog,cname,ty list,ty,nat,nat,instr list,ex-table,ty}_m \right] \Rightarrow \text{bool} \]

**where**

\[ wt-method \ P \ C \ Ts \ Tr mxs mxl 0 \text{ is } xt \ (\Phi C M) \]

— A program is welltyped if it is wellformed and all methods are welltyped

**definition**

\[ wt-jvm-prog :: \text{jvm-prog} \Rightarrow \text{bool} \]

**where**

\[ \text{wf-jvm-prog } \ P \equiv \exists \Phi. \ \text{wf-jvm-prog}_\Phi \ P \]

**lemma**

\[ \text{wt-jvm-progD} : \] \[ \text{wf-jvm-prog}_\Phi \ P \implies \exists \text{wt. } \text{wf-jvm-prog wt } P \]

**lemma**

\[ \text{wt-jvm-prog-impl-wt-instr} : \]

\[ \text{[ wf-jvm-prog}_\Phi \ P ; \]

\[ P \vdash \text{C sees } M:Ts \rightarrow T = (mxs,mxl_0,\text{ins,xt}) \text{ in } C; \ pc < \text{size ins } \]

\[ \implies P,T,mxs,\text{size ins,xt} \vdash \text{ins!pc,pc :: } \Phi C M \]

**lemma**

\[ \text{wt-jvm-prog-impl-wt-start} : \]

\[ \text{[ wf-jvm-prog}_\Phi \ P ; \]

\[ P \vdash \text{C sees } M:Ts \rightarrow T = (mxs,mxl_0,\text{ins,xt}) \text{ in } C \] \[ \implies 0 < \text{size ins } \land \text{wt-start } P \ C \ Ts \ mxl_0 \ (\Phi C M) \]
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 ⇒ ty
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

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 :: ty 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 ⇒ Suc (n − (n − b)) ≤ 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 ⊑ Product. le (op =) r ys ⇒ set xs ⊑ set ys
declare is-relevant-entry-def [simp del]

lemma conjI2: [ A; A ⇒ B ] ⇒ A ∧ B by blast

lemma (in JVM-sl) eff-mono:
  [[wf-prog p P; pc < length is; s ⊑ sup-state-opt P t; app pc t]
  ⇒ set (eff pc s) ⊑ sup-state-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 mzl0 is xt τ s =
  (is ≠ [] ∧
  size τ s = size is ∧
  OK ' set τ s ⊆ states P mxs mzl ∧
  wt-start P C' Ts mzl0 τ s ∧
  wt-app-eff (sup-state-opt P) app eff τ s)

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 ⇒ err list ⇒ ty ⇒ 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 (λP 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
  where
    lbvjvm P mxs maxr T ret 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 ret 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 et mxs mxl 0 cert ins
      ≡ check-cert P mxs (1+size Ts+mxl) cert ∧
      0 < size ins ∧
      (let start = Some ([],(OK (Class C)))#((map OK Ts))@((map replicate mxl) Err));
      result = lbvjvm P mxs (1+size Ts+mxl) T r et cert ins (OK start)
      in result ≠ Err

  definition wt-jvm-prog-lbv :: jvm-prog ⇒ prog-cert ⇒ bool
  where
    wt-jvm-prog-lbv P cert
      ≡ wf-prog (λP C (mn,Ts,T r,(mxs,mxl,b,et)). wt-lbv P C Ts T r mxs mxl et ins (OK 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 ins phi
      ≡ make-cert (exec P mxs T r et ins (map OK phi)) (OK None)

  definition prg-cert :: jvm-prog ⇒ ty ⇒ prog-cert
  where
    prg-cert phi C mn ≡ let (C,Ts,T r,(mxs,mxl,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 ∃rs ∈ list (size is) A. wt-step r Err step rs ∧ OK first ≲ r sl 0
lemma (in start-context) wt-lbv-wt-method:
assumes lbv: wt-lbv P C Ts T, mxs mxl0 xt cert is
shows ∃ τ. wt-method P C Ts T, mxs mxl0 is xt τ

lemma (in start-context) wt-method-wt-lbv:
assumes wt: wt-method P C Ts T, mxs mxl0 is xt τ
defines [simp]: cert ≡ mk-cert P mxs T, xt is τ
shows wt-lbv P C Ts T, mxs mxl0 xt cert is

theorem jvm-lbv-correct:
wjvm-prog-lbv P Cert =⇒ wf-jvm-prog P

theorem jvm-lbv-complete:
assumes wt: wf-jvm-prog Φ P
shows wt-jvm-prog-lbv P (prg-cert P Φ)

4.21 BV Type Safety Invariant

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

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

notation (xsymbols)
confT (−− |− − :≤ − |− −) is [51,51,51,51] 50

abbreviation
confTs :: 'c prog ⇒ heap ⇒ val list ⇒ ty list ⇒ bool
where
(P, h |− vs :≤ T) Ts ≡ list-all2 (confT P h) vs Ts

notation (xsymbols)
confTs (−− |− − :≤ − |− −) is [51,51,51,51] 50

definition conf-f :: jvm-prog ⇒ heap ⇒ ty_i ⇒ bytecode ⇒ frame ⇒ bool
where
conf-f P h ≡ λ(ST, LT) is (stk,loc,C,M,pc).
(P, h ⊢ stk :≤ ST) ∧ P, h ⊢ loc :≤ LT ∧ pc < size is

lemma conf-f-def2:
conf-f P h (ST, LT) is (stk,loc,C,M,pc) ≡
P, h ⊢ stk :≤ ST ∧ P, h ⊢ loc :≤ LT ∧ pc < size is
by (simp add: conf-f-def)

primrec conf-fs : [jvm-prog,heap,ty_p,mname,nat,ty,frame list] ⇒ bool
where
Theory BVConform

\begin{align*}
\text{conf-fs } P \Phi M_0 n_0 T_0 [] &= \text{True} \\
| \text{conf-fs } P \Phi M_0 n_0 T_0 (f \# \text{frs}) &= \\
\text{(let } (\text{stk,loc,C,M,pc}) = f \text{ in} \\
(\exists ST LT Ts T \text{mxs mxl}0 \text{is xt}. \\
\Phi C M \mid pc = \text{Some } (ST,LT) \land \\
(P \vdash C \text{ sees } M \text{:Ts }\rightarrow T = (\text{mxs,mxl}0,\text{is,xt} \text{ in } C) \land \\
(\exists D Ts' T' \text{ m D}'. \\
\text{is!pc = } (\text{Invoke } M_0 n_0) \land ST!n_0 = \text{Class } D \land \\
P \vdash D \text{ sees } M_0;Ts' \rightarrow T' = m \text{ in } D' \land P \vdash T_0 \leq T'_0) \land \\
\text{conf-f P h } (ST, LT) \text{ is f } \land \text{conf-fs P h } \Phi M (\text{size Ts} T \text{frs}) )
\end{align*}

\text{definition correct-state } :: [\text{jvm-prog,tyP,jvm-state}] \Rightarrow \text{bool} \\
\text{(\text{-,- } \mid - [\text{ok}] [61,0,0] 61)}

\text{where}

\text{correct-state } P \Phi \equiv \lambda (xp, h, frs).

\text{case xp of}
\text{None } \Rightarrow (\text{case frs of}
\text{[]} \Rightarrow \text{True} \\
(f \# \text{fs} ) \Rightarrow P \vdash h^\sqrt{\land} \\
\text{(let } (\text{stk,loc,C,M,pc}) = f \text{ in} \exists Ts T \text{mxs mxl}0 \text{is xt}. \\
(P \vdash C \text{ sees } M \text{:Ts }\rightarrow T = (\text{mxs,mxl}0,\text{is,xt} \text{ in } C) \land \\
\Phi C M \mid pc = \text{Some } \tau \land \\
\text{conf-f P h } \tau \text{ is f } \land \text{conf-fs P h } \Phi M (\text{size Ts} T \text{frs}))
\text{| Some x } \Rightarrow \text{frs = []}

\text{notation } (\text{xsymbols})
\text{correct-state } (\text{-,- } \vdash \sqrt{61,0,0} 61)

\subsection{4.21.1 Values and \top}

\text{lemma confT-Err [iff]: } P, h \vdash x : \leq \top \text{ Err} \\
\text{by (simp add: confT-def)}

\text{lemma confT-OK [iff]: } P, h \vdash x : \leq \top \text{ OK } T = (P, h \vdash x : \leq T) \\
\text{by (simp add: confT-def)}

\text{lemma confT-cases:}
\text{P, h \vdash x : } \leq \top \text{ X = (X = Err } \lor (\exists T. X = \text{OK } T \land P, h \vdash x : \leq T)) \\
\text{by (cases X) auto}

\text{lemma confT-hext [intro?, trans]:}
\text{[ P, h \vdash x : } \leq \top \text{ T; h } \leq h' ] \Rightarrow P, h' \vdash x : \leq \top T \\
\text{by (cases } T \text{) (blast intro: conf-hext)+}

\text{lemma confT-widen [intro?, trans]:}
\text{[ P, h \vdash x : } \leq \top \text{ T; P } \vdash T \leq \top T' ] \Rightarrow P, h \vdash x : \leq \top T' \\
\text{by (cases } T \text{, auto intro: conf-widen)}

\subsection{4.21.2 Stack and Registers}

\text{lemmas confTs-Cons1 [iff] } = \text{list-all2-Cons1 [of confT P h] for } P h
lemma confTs-confT-sup:
\[ P, h \vdash \text{loc} : [\leq \top] \ LT \implies P, h \vdash \text{loc} n : [\leq T'] \]
\[ \implies P, h \vdash (\text{loc} n) : [\leq T'] \]

lemma confTs-hext [intro?]:
\[ P, h \vdash \text{loc} : [\leq \top] \ LT \implies h \leq h' \implies P, h' \vdash \text{loc} : [\leq \top] \ LT \]
by (fast elim: list-all2-mono confT-hext)

lemma confTs-widen [intro?, trans]:
\[ P, h \vdash \text{loc} : [\leq \top] \ LT \implies P \vdash \text{LT} : [\leq \top] \ LT' \implies P, h \vdash \text{loc} : [\leq \top] \ LT' \]
by (rule list-all2-trans, rule confT-widen)

lemma confTs-map [iff]:
\[ \forall vs. (P, h \vdash vs : [\leq \top] \text{map OK Ts}) = (P, h \vdash vs : [\leq \top] Ts) \]
by (induct Ts) (auto simp add: list-all2-Cons2)

lemma reg-widen-Err [iff]:
\[ \forall LT. (P \vdash \text{replicate n_Err} : [\leq \top] \ LT) = (LT = \text{replicate n_Err}) \]
by (induct n) (auto simp add: list-all2-Cons1)

lemma confTs-Err [iff]:
\[ P, h \vdash \text{replicate n_v} : [\leq \top] \text{replicate n_Err} \]
by (induct n) auto

4.21.3 correct-frames
lemmas [simp del] = fun-upd-apply

lemma conf-fs-hext:
\[ \forall M n T_r.
\[ \exists \Phi M n T_r \text{frs}; h \leq h' \] \implies \text{conf-fs} P h \Phi M n T_r \text{frs} \]
end

4.22 BV Type Safety Proof

thory 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:
\[ \text{match-ex-table} P C \text{pc} \text{xt} = \text{Some} (\text{pc}', \text{d}') \implies \]
We can prove separately that the recursive search for exception handlers (\textit{find-handler}) in the frame stack results in a conforming state (if there was no matching exception handler in the current frame). We require that the exception is a valid heap address, and that the state before the exception occurred conforms.

\textbf{term find-handler}

\textbf{lemma uncaught-xcpt-correct:}
\begin{align*}
\text{assumes } & \forall t. P \vdash (\text{find-handler } P \ xcp \ h) \ \\
\text{assumes } & \exists h. xcp = \text{Some } obj \\
\text{shows } & \exists h. \ xcp = \text{Some } obj \\
\text{(is } & \exists h. xcp = \text{Some } obj) \\
\text{by } & \text{(induct } h) \\
\end{align*}

\textbf{lemma conf-sys-xcpts:}
\begin{align*}
\text{\sout{\text{preallocated } h; C \in \text{sys-xcpts}} } & \implies P,h \vdash \text{Addr (addr-of-sys-xcpt } C) : \leq \text{Class } C \\
\text{by } & \text{(auto elim: preallocatedE)} \\
\end{align*}

\textbf{lemma match-ex-table-SomeD:}
\begin{align*}
\forall t. P \ xcp = \text{Some } (pc',d') \implies \exists t. \text{matches-ex-entry } P \ C \ xcp = \text{Some } (pc',d') \\
\text{by } & \text{(induct } t) \\
\end{align*}

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

\textbf{lemma xcp-correct:}
\begin{align*}
\text{fixes } & \sigma' :: \text{jvm-state} \\
\text{assumes } & \Sigma' :: \text{jvm-state} \\
\text{assumes } & \text{wtp: } \text{uf-jvm-prog}_P \ P \\
\text{assumes } & \text{meth: } P \vdash C \text{ sees } M.Ts \rightarrow T := (mxs,mxl_{0},\text{ins},xt) \text{ in } C \\
\text{assumes } & \text{wt: } P,T,mxs,\text{size ins},xt \vdash \text{ins},pc,pc :: \Phi \ C M \\
\text{assumes } & \text{xpc: } \text{fist (exec-instr (ins!pc) P h stk loc C M pc frs) = Some } xcp \\
\text{assumes } & (\text{s'': Some } \sigma' = \text{exec (P, None, h, (stk,loc,C,M,pc)#frs)}) \\
\text{assumes } & \text{correct: } P,\Phi \vdash (\text{None, h, (stk,loc,C,M,pc)#frs}) \ \\
\text{shows } & P,\Phi \vdash \sigma' \ \\
\end{align*}

\subsection*{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.

\textbf{declare defS1 [simp]}

\textbf{lemma Invoke-correct:}
\begin{align*}
\text{fixes } & \sigma' :: \text{jvm-state} \\
\end{align*}
assumes wt-prog: \( wf:jvm-prog \Phi P \)
assumes meth-C: \( P \vdash C \text{ sees } M:T \rightarrow T = (mxs,mxl_0,ins,xt) \text{ in } C \)
assumes ins: \( \text{ ins ! pc = } \text{ Invoke } M' \ n \)
assumes wtii: \( P,T,mxs,\text{ size ins,xt } \vdash \text{ ins!pc,pc :: } \Phi C M \)
assumes \( \sigma' \): Some \( \sigma' = \text{ exec } (P, \text{ None, } h, (stk,loc,C,M,pc)\#frs) \)
assumes approx: \( P,\Phi \vdash (\text{ None, } h, (stk,loc,C,M,pc)\#frs) \checkmark \)
assumes no-xcp: \( \text{ fst (exec-instr } (\text{ ins!pc }) \ P \ h \text{ stk loc } C M \ p c \ f r s) = \text{ None } \)
shows \( P,\Phi \vdash \sigma'\checkmark \)
declare list-all2-Cons2 [iff]

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

shows \( P,\Phi \vdash \sigma'\checkmark \)
declare sup-state-opt-any-Any [iff]
declare not-Err-eq [iff]

lemma Load-correct:
[ wt-prog \( \text{ wt P; } \)
\( P \vdash C \text{ sees } M:T \rightarrow T = (mxs,mxl_0,ins,xt) \text{ in } C; \)
\( \text{ ins!pc = Load } \text{ idx}; \)
\( P,T,mxs,\text{ size ins,xt } \vdash \text{ ins!pc,pc :: } \Phi C M; \)
Some \( \sigma' = \text{ exec } (P, \text{ None, } h, (stk,loc,C,M,pc)\#frs) \)
\( P,\Phi \vdash (\text{ None, } h, (stk,loc,C,M,pc)\#frs) \checkmark \) ]
\( \Rightarrow P,\Phi \vdash \sigma'\checkmark \)
by (fastforce dest: sees-method-fun [of - C] elim!: confTs-confT-sup)
declare [[simproc del: list-to-set-comprehension]]

lemma Store-correct:
[ wt-prog \( \text{ wt P; } \)
\( P \vdash C \text{ sees } M:T \rightarrow T = (mxs,mxl_0,ins,xt) \text{ in } C; \)
\( \text{ ins!pc = Store } \text{ idx}; \)
\( P,T,mxs,\text{ size ins,xt } \vdash \text{ ins!pc,pc :: } \Phi C M; \)
Some \( \sigma' = \text{ exec } (P, \text{ None, } h, (stk,loc,C,M,pc)\#frs) \)
\( P,\Phi \vdash (\text{ None, } h, (stk,loc,C,M,pc)\#frs) \checkmark \) ]
\( \Rightarrow P,\Phi \vdash \sigma'\checkmark \)

lemma Push-correct:
[ wt-prog \( \text{ wt P; } \)
\( P \vdash C \text{ sees } M:T \rightarrow T = (mxs,mxl_0,ins,xt) \text{ in } C; \)
\( \text{ ins!pc = Push } v; \)
\( P,T,mxs,\text{ size ins,xt } \vdash \text{ ins!pc,pc :: } \Phi C M; \)
Some \( \sigma' = \text{ exec } (P, \text{ None, } h, (stk,loc,C,M,pc)\#frs) \)
\( P,\Phi \vdash (\text{ None, } h, (stk,loc,C,M,pc)\#frs) \checkmark \) ]
\( \Rightarrow P,\Phi \vdash \sigma'\checkmark \)

lemma Cast-conf2:
lemma Checkcast-correct:
\[
\text{wf-jvm-prog} \ P; \ P.h \vdash v :\leq T; \ \text{is-refT} \ T; \ \text{cast-ok} \ P \ C \ h \ v;
\]
\[
P \vdash \text{Class} \ C :\leq T'; \ \text{is-class} \ P \ C
\]
\[
\implies P.h \vdash v :\leq T'
\]

declare split-paired-All \ simp \ del

lemmas widens-Cons \ iff = list-all2-Cons1 \ [of widen \ P] \ for \ P

lemma Getfield-correct:
\begingroup
fixes \(\sigma'\) :: \jvm-state
assumes \(\text{wf}\) : \(\text{wf-prog} \ \text{wt} \ P\)
assumes \(mC\) : \(P \vdash \text{C sees} \ M : T \rightarrow (mxs,mzl_0,ins,xt) \ \text{in} \ C\)
assumes \(i\) : \(\text{ins!pc} = \text{Getfield} \ F \ D\)
assumes \(wt\) : \(P,T,mxs,size \ ins,xt \vdash \text{ins!pc,pc} :: \Phi \ C \ M\)
assumes \(s'\) : \(\text{Some} \ \sigma' \ = \ \text{exec} \ (P,None,h,(stk,loc,C,M,pc)\#frs)\)
assumes \(cf\) : \(P,\Phi \vdash (\text{None},h,(stk,loc,C,M,pc)\#frs)\sqrt{\}
assumes \(xc\) : \(\text{fst} \ (\text{exec-instr} \ (\text{ins!pc}) \ P \ stk \ loc \ C \ M \ pc \ frs) = \text{None} \)
\endgroup

shows \(P,\Phi \vdash \sigma'\sqrt{\}

lemma Putfield-correct:
\begingroup
fixes \(\sigma'\) :: \jvm-state
assumes \(\text{wf}\) : \(\text{wf-prog} \ \text{wt} \ P\)
assumes \(mC\) : \(P \vdash \text{C sees} \ M : T \rightarrow (mxs,mzl_0,ins,xt) \ \text{in} \ C\)
assumes \(i\) : \(\text{ins!pc} = \text{Putfield} \ F \ D\)
assumes \(wt\) : \(P,T,mxs,size \ ins,xt \vdash \text{ins!pc,pc} :: \Phi \ C \ M\)
assumes \(s'\) : \(\text{Some} \ \sigma' \ = \ \text{exec} \ (P,None,h,(stk,loc,C,M,pc)\#frs)\)
assumes \(cf\) : \(P,\Phi \vdash (\text{None},h,(stk,loc,C,M,pc)\#frs)\sqrt{\}
assumes \(xc\) : \(\text{fst} \ (\text{exec-instr} \ (\text{ins!pc}) \ P \ stk \ loc \ C \ M \ pc \ frs) = \text{None} \)
\endgroup

shows \(P,\Phi \vdash \sigma'\sqrt{\}

lemma has-fields-b-fields:
\(P \vdash \text{C has-fields} \ FDTs \implies \text{fields} \ P \ C = FDTs\)

lemma oconf-blank [intro, simp]:
\[\text{is-class} \ P \ C; \ \text{wf-prog} \ \text{wt} \ P \implies P.h \vdash \text{blank} \ P \ C \sqrt{\}

lemma obj-ty-blank [iff]: \obj-ty \ (\text{blank} \ P \ C) = \text{Class} \ C
by \(\text{simp add: blank-def}\)

lemma New-correct:
\begingroup
fixes \(\sigma'\) :: \jvm-state
assumes \(\text{wf}\) : \(\text{wf-prog} \ \text{wt} \ P\)
assumes \(\text{meth}\) : \(P \vdash \text{C sees} \ M : T \rightarrow (mxs,mzl_0,ins,xt) \ \text{in} \ C\)
assumes \(i\) : \(\text{ins!pc} = \text{New} \ X\)
assumes \(wt\) : \(P,T,mxs,size \ ins,xt \vdash \text{ins!pc,pc} :: \Phi \ C \ M\)
\endgroup
\begin{enumerate}
\item \textbf{assumes exec:} Some $\sigma' = \text{exec} (P, \text{None}, h, (\text{stk}, \text{loc}, C, M, \text{pc})\#\text{frs})$
\item \textbf{assumes conf:} $P, \Phi \vdash (\text{None}, h, (\text{stk}, \text{loc}, C, M, \text{pc})\#\text{frs}) \checkmark$
\item \textbf{assumes no-x:} \text{fst} (\text{exec-inst} (\text{ins!pc}) P h \text{ stk loc} C M \text{ pc frs}) = \text{None}$
\item \textbf{shows} $P, \Phi \vdash \sigma' \checkmark$
\end{enumerate}

\textbf{lemma Goto-correct:}

\begin{enumerate}
\item \textbf{wf-prog wt P;}
\item $P \vdash C \text{ sees M:Ts} \Rightarrow T = (\text{mzl}_{0}, \text{ins}, \text{xt}) \text{ in C}$;
\item \text{ins! pc} = \text{Goto branch};
\item $P, T, \text{mzl}_{0}, \text{size ins}, \text{xt} \vdash \text{ins!pc} : C M$;
\item Some $\sigma' = \text{exec} (P, \text{None}, h, (\text{stk}, \text{loc}, C, M, \text{pc})\#\text{frs})$; \text{frs}$\checkmark$
\item $P, \Phi \vdash (\text{None}, h, (\text{stk}, \text{loc}, C, M, \text{pc})\#\text{frs}) \Rightarrow P, \Phi \vdash \sigma' \checkmark$
\end{enumerate}

\textbf{lemma IIfFalse-correct:}

\begin{enumerate}
\item \textbf{wf-prog wt P;}
\item $P \vdash C \text{ sees M:Ts} \Rightarrow T = (\text{mzl}_{0}, \text{ins}, \text{xt}) \text{ in C}$;
\item \text{ins! pc} = \text{IIfFalse branch};
\item $P, T, \text{mzl}_{0}, \text{size ins}, \text{xt} \vdash \text{ins!pc} : C M$;
\item Some $\sigma' = \text{exec} (P, \text{None}, h, (\text{stk}, \text{loc}, C, M, \text{pc})\#\text{frs})$; \text{frs}$\checkmark$
\item $P, \Phi \vdash (\text{None}, h, (\text{stk}, \text{loc}, C, M, \text{pc})\#\text{frs}) \Rightarrow P, \Phi \vdash \sigma' \checkmark$
\end{enumerate}

\textbf{lemma CmpEq-correct:}

\begin{enumerate}
\item \textbf{wf-prog wt P;}
\item $P \vdash C \text{ sees M:Ts} \Rightarrow T = (\text{mzl}_{0}, \text{ins}, \text{xt}) \text{ in C}$;
\item \text{ins! pc} = \text{CmpEq};
\item $P, T, \text{mzl}_{0}, \text{size ins}, \text{xt} \vdash \text{ins!pc} : C M$;
\item Some $\sigma' = \text{exec} (P, \text{None}, h, (\text{stk}, \text{loc}, C, M, \text{pc})\#\text{frs})$; \text{frs}$\checkmark$
\item $P, \Phi \vdash (\text{None}, h, (\text{stk}, \text{loc}, C, M, \text{pc})\#\text{frs}) \Rightarrow P, \Phi \vdash \sigma' \checkmark$
\end{enumerate}

\textbf{lemma Pop-correct:}

\begin{enumerate}
\item \textbf{wf-prog wt P;}
\item $P \vdash C \text{ sees M:Ts} \Rightarrow T = (\text{mzl}_{0}, \text{ins}, \text{xt}) \text{ in C}$;
\item \text{ins! pc} = \text{Pop};
\item $P, T, \text{mzl}_{0}, \text{size ins}, \text{xt} \vdash \text{ins!pc} : C M$;
\item Some $\sigma' = \text{exec} (P, \text{None}, h, (\text{stk}, \text{loc}, C, M, \text{pc})\#\text{frs})$; \text{frs}$\checkmark$
\item $P, \Phi \vdash (\text{None}, h, (\text{stk}, \text{loc}, C, M, \text{pc})\#\text{frs}) \Rightarrow P, \Phi \vdash \sigma' \checkmark$
\end{enumerate}

\textbf{lemma IAdd-correct:}

\begin{enumerate}
\item \textbf{wf-prog wt P;}
\item $P \vdash C \text{ sees M:Ts} \Rightarrow T = (\text{mzl}_{0}, \text{ins}, \text{xt}) \text{ in C}$;
\item \text{ins! pc} = \text{IAdd};
\item $P, T, \text{mzl}_{0}, \text{size ins}, \text{xt} \vdash \text{ins!pc} : C M$;
\item Some $\sigma' = \text{exec} (P, \text{None}, h, (\text{stk}, \text{loc}, C, M, \text{pc})\#\text{frs})$; \text{frs}$\checkmark$
\item $P, \Phi \vdash (\text{None}, h, (\text{stk}, \text{loc}, C, M, \text{pc})\#\text{frs}) \Rightarrow P, \Phi \vdash \sigma' \checkmark$
\end{enumerate}

\textbf{lemma Throw-correct:}

\begin{enumerate}
\item \textbf{wf-prog wt P;}
\item $P \vdash C \text{ sees M:Ts} \Rightarrow T = (\text{mzl}_{0}, \text{ins}, \text{xt}) \text{ in C}$;
\item \text{ins! pc} = \text{Throw};
\item Some $\sigma' = \text{exec} (P, \text{None}, h, (\text{stk}, \text{loc}, C, M, \text{pc})\#\text{frs})$; \text{frs}$\checkmark$
\item $P, \Phi \vdash (\text{None}, h, (\text{stk}, \text{loc}, C, M, \text{pc})\#\text{frs}) \Rightarrow P, \Phi \vdash \sigma' \checkmark$
\end{enumerate}
fst (exec-instr (ins!pc) P h stk loc C M pc frs) = None ]
⇒ P,Φ ⊢ σ'√
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 ∈ ∈ ]
P ⊢ C sees M:Ts→T=(mxs,mxl0,ins,xt) in C;
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'√

4.22.4 Main

**lemma correct-state-impl-Some-method:**

P,Φ ⊢ (None, h, (stk,loc,C,M,pc)#frs)√
⇒ ∃m Ts T. P ⊢ C sees M:Ts→T = m in C
by fastforce

**lemma BV-correct-1 [rule-format]:**

∀σ. [ wf-jvm-prog ∈ ∈ ]; P,Φ ⊢ σ√ ]
⇒ P ⊢ σ→jvm→1 σ' ─→ P,Φ ⊢ σ'√

**theorem progress:**

[ xp=None; frs≠[] ]
⇒ ∃σ'. P ⊢ (xp,h,frs)→jvm→1 σ'√
by (clarsimp simp add: exec-1-iff neq-Nil-conv split-beta
simp del: split-paired-Ex)

**lemma progress-conform:**

[ wf-jvm-prog ∈ ∈ ]; P ⊢ σ→jvm→1 σ'√
⇒ P,Φ ⊢ σ√ ]
⇒ ∃σ'. P ⊢ (xp,h,frs)→jvm→1 σ' ∧ P,Φ ⊢ σ'√

**theorem BV-correct [rule-format]:**

[ wf-jvm-prog ∈ ∈ ]; P ⊢ σ→jvm→1 σ'√
⇒ P,Φ ⊢ σ√ ]
⇒ ∃σ'. P ⊢ (xp,h,frs)→jvm→1 σ' ∧ P,Φ ⊢ σ'√

**lemma bconf-start:**

assumes wf: wf-prog wf-mb P
shows P ⊢ (start-heap P)√

**lemma BV-correct-initial:**

assumes welltyped: wf-jvm-prog P
shows P ⊢ (start-state P)√

**theorem typesafe:**

assumes main-method: P ⊢ C sees M:[]→T = m in C
shows P ⊢ start-state P C M √
end

4.23 Welltyped Programs produce no Type Errors
lemma \textit{has-methodI}:
\[ P \vdash C \text{ sees } M : \mathsf{Ts} \rightarrow T = m \text{ in } D \implies P \vdash C \text{ has } M \]
by (unfold \textit{has-method-def}) blast

Some simple lemmas about the type testing functions of the defensive JVM:

lemma \textit{typeof-NoneD} [simp, dest]: \textit{typeof } v = \textit{Some } x = \implies \neg \textit{is-Addr } v
by (cases \textit{v}) auto

lemma \textit{is-Ref-def2}:
\[ \textit{is-Ref } v = (v = \text{Null } \lor (\exists \ a. \ v = \text{Addr } a)) \]
by (cases \textit{v}) (auto simp add: \textit{is-Ref-def})

lemma [iff]: \textit{is-Ref Null} by (simp add: \textit{is-Ref-def2})

lemma \textit{is-IntgI} [intro, simp]: \[ P, h \vdash v : \leq \mathsf{T} = \implies \textit{is-refT } \mathsf{T} = \implies \textit{is-Intg } v \]

lemma \textit{is-BoolI} [intro, simp]: \[ P, h \vdash v : \leq \mathsf{Boolean} = \implies \textit{is-Bool } v \]

declare \textit{defs1} [simp del]

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

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

\textbf{theorem} \textit{no-type-error}:
\begin{align*}
\text{fixes } \sigma :: \mathsf{jvm-state} \\
\text{assumes welltyped}: \textit{wf-jvm-prog}\ P \text{ and } \textit{conforms}: P, \Phi \vdash \sigma \checkmark \\
\text{shows exec-d } P \sigma \neq \textit{TypeError} \\
\end{align*}

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

\textbf{theorem} \textit{welltyped-aggressive-imp-defensive}:
\begin{align*}
\textit{wf-jvm-prog}\ P \implies P, \Phi \vdash \sigma \checkmark \implies P \vdash \sigma -\textit{jvm} \rightarrow \sigma' \\
\implies P \vdash (\textit{Normal } \sigma) -\textit{jvmd} \rightarrow (\textit{Normal } \sigma') \\
\end{align*}

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).

\textbf{corollary} \textit{welltyped-commutes}:
\begin{align*}
\text{fixes } \sigma :: \mathsf{jvm-state} \\
\text{assumes \textit{wf}}: \textit{wf-jvm-prog}\ P \text{ and } \textit{conforms}: P, \Phi \vdash \sigma \checkmark \\
\text{shows } P \vdash (\textit{Normal } \sigma) -\textit{jvm} \rightarrow (\textit{Normal } \sigma') \\
\text{apply rule} \\
\text{apply (erule defensive-imp-aggressive)} \\
\text{apply (erule welltyped-aggressive-imp-defensive [OF \textit{wf conforms}])} \\
\text{done} \\
\end{align*}

\textbf{corollary} \textit{welltyped-initial-commutes}:
\begin{align*}
\text{assumes \textit{wf}}: \textit{wf-jvm-prog}\ P \\
\text{assumes \textit{meth}}: P \vdash C \text{ sees } M :: \rightarrow T = b \text{ in } C \\
\text{defines start: } \sigma \equiv \text{start-state } P \ C \ M \\
\text{shows } P \vdash (\textit{Normal } \sigma) -\textit{jvmd} \rightarrow (\textit{Normal } \sigma') = P \vdash \sigma -\textit{jvm} \rightarrow \sigma' \\
\text{proof} -
from \textit{wf} obtain \( \Phi \) where \( \textit{wf} : \textit{wf-jvm-prog}_\Phi \) \( P \) by (\texttt{auto simp: \textit{wf-jvm-prog-def}})

from \textit{this meth} have \( P, \Phi \vdash \sigma \checkmark \) unfolding \texttt{start} by (\texttt{rule \textit{BV-correct-initial}})

with \( \textit{wf}' \) show \( \textit{thesis} \) by (\texttt{rule welltyped-commutes})

qed

\begin{isabelle}
\textbf{lemma not-TypeError-eq [iff]:} \\
x \neq \text{TypeError} = (\exists t. x = \text{Normal } t) \\
by (cases x) \texttt{auto}
\end{isabelle}

\begin{isabelle}
\textbf{locale cnf =} \\
fixes \( P \) and \( \Phi \) and \( \sigma \) \\
assumes \textit{wf}: \textit{wf-jvm-prog}_\Phi \ P \\
assumes \textit{cnf}: \textit{correct-state } P \ \Phi \ \sigma
\end{isabelle}

\begin{isabelle}
\textbf{theorem (in \textit{cnf}) no-type-errors:} \\
P \vdash (\text{Normal } \sigma) - \text{jvmd} \rightarrow \sigma' \rightarrow \sigma' \neq \text{TypeError} \\
apply (\texttt{unfold exec-all-d-def1}) \\
apply (\texttt{erule rtrancl-induct}) \\
\quad apply \ (\texttt{simp}) \\
apply (\texttt{fold exec-all-d-def1}) \\
apply (\texttt{insert cnf \textit{wf}}) \\
apply \ (\texttt{clarsimp}) \\
apply (\texttt{drule defensive-imp-aggressive}) \\
apply (\texttt{frule (2) \textit{BV-correct}}) \\
apply (\texttt{drule (1) no-type-error}) \texttt{back} \\
apply (\texttt{auto simp add: exec-1-d-eq}) \\
\texttt{done}
\end{isabelle}

\begin{isabelle}
\textbf{locale start =} \\
fixes \( P \) and \( C \) and \( M \) and \( \sigma \) and \( T \) and \( b \) \\
assumes \textit{wf}: \textit{wf-jvm-prog} \ P \\
assumes \textit{sees}: \ P \vdash C \ \text{sees} M : []\rightarrow T = b \ \text{in } C \\
defines \sigma \equiv \text{Normal } (\text{start-state } P \ C \ M)
\end{isabelle}

\begin{isabelle}
\textbf{corollary (in \textit{start}) bv-no-type-error:} \\
\textbf{shows} \ P \vdash \sigma - \text{jvmd} \rightarrow \sigma' \rightarrow \sigma' \neq \text{TypeError} \\
\textbf{proof} - \\
from \textit{wf} obtain \( \Phi \) where \( \textit{wf-jvm-prog}_\Phi \) \( P \) by (\texttt{auto simp: \textit{wf-jvm-prog-def}}) \\
moreover \\
with \textit{sees} have \textit{correct-state } P \ \Phi \ (\text{start-state } P \ C \ M) \\
by - (\texttt{rule \textit{BV-correct-initial}}) \\
ultimately have \textit{cnf} \ P \ \Phi \ (\text{start-state } P \ C \ M) \ \texttt{by (rule cnf.intro)} \\
moreover assume \( P \vdash \sigma - \text{jvmd} \rightarrow \sigma' \) \\
ultimately show \( \textit{thesis} \) by (\texttt{unfold \sigma-def}) (\texttt{rule \textit{cnf.no-type-errors}}) \\
\texttt{qed}
\end{isabelle}

end

\section{Example Welltypings}

theory \textit{BVExample}
This theory shows type correctness of the example program in section 3.7 (p. 76) 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' [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 relation spelled out:

lemma subcls1:
subcls1 E = { (list-name, Object), (test-name, Object), (NullPointer, Object),
              (ClassCast, Object), (OutOfMemory, Object) }

The subclass relation is acyclic; hence its converse is well founded:

lemma notin-rtrancl:
(a,b) ∈ r∗ ⟹ a ≠ b ⟹ (∀ y. (a,y) ∉ r) ⟹ False
by (auto elim: converse-rtranclE)

lemma acyclic-subcls1-E: acyclic (subcls1 E)
lemma wf-subcls1-E: wf ((subcls1 E)−1)

Method and field lookup:

lemma method-append [simp]:
method E list-name append-name =
    (list-name, [Class list-name], Void, 3, 0, append-ins, [(1, 2, NullPointer, 7, 0)])

lemma method-makelist [simp]:
method E test-name makelist-name =
    (test-name, [], Void, 3, 2, make-list-ins, [])

lemma field-val [simp]:
field E list-name val-name = (list-name, Integer)

lemma field-next [simp]:
field E list-name next-name = (list-name, Class list-name)

lemma [simp]: fields E Object = []
by (fastforce intro: fields-def2 Fields.intros)

lemma [simp]: fields E NullPointer = []
by (fastforce simp add: distinct-classes intro: fields-def2 Fields.intros)

lemma [simp]: fields E ClassCast = []
by (fastforce simp add: distinct-classes intro: fields-def2 Fields.intros)

lemma [simp]: fields E OutOfMemory = []
by (fastforce simp add: distinct-classes intro: fields-def2 Fields.intros)
lemma [simp]: fields E test-name = []
lemmas [simp] = is-class-def

4.24.2 Program structure

The program is structurally wellformed:

lemma wf-struct:
    wf-prog (λG 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]),
( [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]),
( [Class list-name, Class list-name], [Class list-name, Class list-name]),

The next definition and three proof rules implement an algorithm to enumerate natural numbers. The command apply (elim pc-end pc-next pc-0) transforms a goal of the form

pc < n ⇒ P pc

into a series of goals

P (θ::'a)

P (Suc 0)

...
Theory BVExample

definition intervall :: nat ⇒ nat ⇒ nat ⇒ bool (- ∈ [-, -])
where
  \( x \in [a, b) \equiv a \leq x \land x < b \)

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

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

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

Some abbreviations for readability

abbreviation Clist ≡ Class list-name
abbreviation Ctest ≡ Class test-name

definition phi-makelist :: \( ty_m (\varphi_m) \)
where
  \( \varphi_m \equiv \text{map } (\lambda(x,y). \text{Some } (x, y))\)

lemma types-makelist [simp]: check-types E 3 (Suc (Suc 0)) (map OK \( \varphi_m \))
lemma wt-makelist [simp]:
  wt-method E test-name [] Void 3 2 make-list-ins [] \( \varphi_m \)
lemma \texttt{wf-md'E}: \\
[\lfloor \texttt{wf-prog \ hrmd P;} \\\n\neg C S fs ms m, [(C,S,fs,ms) \in \texttt{set P}; m \in \texttt{set ms}] \implies \texttt{wf-md'' P C m} \rfloor] \implies \texttt{wf-prog \ hrmd'' P}

The whole program is welltyped:

definition \texttt{Phi} :: \texttt{ty P (\Phi)} where \\
\texttt{Phi C mn} \equiv \texttt{if C = test-name \land mn = makelist-name then \varphi_m else} \\
\texttt{if C = list-name \land mn = append-name then \varphi_a else} []

lemma \texttt{wf-prog}: \\
\texttt{wf-jvm-prog \ hrmd \ Phi E}

4.24.4 Conformance

Execution of the program will be typesafe, because its start state conforms to the welltyping:

lemma \texttt{E, \Phi \vdash start-state E test-name makelist-name} √

4.24.5 Example for code generation: inferring method types

definition \texttt{test-kil} :: \texttt{jvm-prog} \\
\Rightarrow \texttt{cname} \\
\Rightarrow \texttt{ty list} \\
\Rightarrow \texttt{ty} \\
\Rightarrow \texttt{nat} \\
\Rightarrow \texttt{nat} \\
\Rightarrow \texttt{ex-table} \\
\Rightarrow \texttt{instr list} \\
\Rightarrow \texttt{ty i' err list} where \\
\texttt{test-kil G C pTs rT mxs mxl et instr} \equiv \\
(let \texttt{first} = \texttt{Some ([], (OK (Class C)) (\# (map OK pTs) (\# (replicate mxl Err))));} \\
\texttt{start} = \texttt{OK first (\# (replicate (size instr - 1) (OK None))}} \\
in \texttt{kiljvm G mxs (1 + size pTs + mxl)} \texttt{rT instr et start})

lemma \texttt{[code]}:
unstables r step ss =
\texttt{fold} (\lambda p A. \texttt{if \neg \text{stable r step ss p} then insert p A else A}) [0..<\text{size ss}] {} 

\textbf{proof} –
\textbf{have} unstables r step ss = (\texttt{UN p:\{..<\text{size ss}\}. if \neg \text{stable r step ss p} then \{p\} else \{\}}) 
\texttt{apply (unfold unstables-def)}
\texttt{apply (rule equalityI)}
\texttt{apply (rule subsetI)}
\texttt{apply (erule CollectE)}
\texttt{apply (erule conjE)}
\texttt{apply (rule UN-I)}
\texttt{apply simp}
\texttt{apply simp}
\texttt{apply (rule subsetI)}
\texttt{apply (erule UN-E)}
\texttt{apply (case_tac \neg \text{stable r step ss p})}
\texttt{apply simp+}
\texttt{done}
\textbf{also have} \texttt{\big\{. (UN p:\{..<\text{size ss}\}. f p) = Union (set (map f [0..<\text{size ss}])) by auto}
\textbf{also note} Sup-set-fold \textbf{also note} fold-map
\textbf{also have} \texttt{op \cup o (\lambda p. if \neg \text{stable r step ss p} then \{p\} else \{\}) =}
\texttt{(\lambda p A. if \neg \text{stable r step ss p} then insert p A else A)}
\texttt{by(auto simp add; fun-eq-iff)}
\textbf{finally show} \texttt{?thesis} .
definition some-elem :: 'a set ⇒ 'a where [code def]:
some-elem = (%S. SOME x. x : S)
code-printing
constant some-elem ⇒ (SML) (case/ - of/ Set/ xs/ =>/ hd/ xs)

This code setup is just a demonstration and not sound!

notepad begin
have some-elem (set [False, True]) = False by eval
moreover have some-elem (set [True, False]) = True by eval
ultimately have False by (simp add: some-elem-def)
end

lemma [code]:
iter f step ss w = while (λ(ss, w). ~ Set.is-empty w)
  (λ(ss, w).
    let p = some-elem w in propa f (step p (ss ! p)) ss (w − {p}))
  (ss, w)
unfolding iter-def Set.is-empty-def some-elem-def ..

lemma JVM-sup-unfold [code]:
JVM-SemiType.sup S m n = lift2 (Opt.sup
  (Product.sup (Listn.sup (SemiType.sup S)))
  (λx y. OK (map2 (lift2 (SemiType.sup S)) x y))))
apply (unfold JVM-SemiType.sup-def JVM-SemiType.sl-def Opt.esl-def Err.sl-def
stk-esl-def loc-sl-def Product.esl-def
Listn.sl-def upto-esl-def SemiType.esl-def Err.esl-def)
by simp

lemmas [code] = SemiType.sup-def [unfolded exec-lub-def] JVM-le-unfold

lemmas [code] = lcsub-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, Tr, (T1#ST, LT)) =
  Predicate.holds (Predicate.bind (sees-field-i-i-o-i P C F C) (λT1. if P ⊨ T ≤ Class C then
  Predicate.single () else bot))
by(auto simp add: Predicate.holds-eq intro: sees-field-i-i-i-o-i elim: sees-field-i-i-i-o-iE)

lemma [code]:
app_i (Putfield F C, P, pc, mxs, Tr, (T1#T2#ST, LT)) =
  Predicate.holds (Predicate.bind (sees-field-i-i-o-i P C F C) (λT1. if P ⊨ T2 ≤ (Class C) ∧ P ⊨ 
  T1 ≤ T1 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-o-iI)
\[\text{app}_n (\text{Invoke } M \ n, \ P, \ pc, \ mxs, \ T_r, \ (ST,LT)) = \]
\[(n < \text{length } ST \land \]
\[(ST)!n \neq NT \rightarrow \]
\[\text{Class } C \Rightarrow \text{Predicate.holds} (\text{Predicate.bind} (\text{Method-i-i-o-o-o-o } P \ C \ M) (\lambda(Ts, \ T, \ m, \ D). \ 
\text{if } P \vdash \text{rev} (\text{take } n \ ST) \leq Ts \text{ then } \text{Predicate.single} () \text{ else } \text{bot}) \]
\[| - \Rightarrow False )]))\]
\[\text{by (fastforce simp add: Predicate.holds-eq simp del: eval-bind split: ty.split-asm split-if-asm intro: bindI Method-i-i-o-o-o-oI elim: bindE Method-i-i-o-o-o-oE)}\]

\text{lemmas [code] =}
\text{SemiType.sup-def [unfolded exec-lub-def]}
\text{widen.equation}
\text{is-relevant-class.simps}

\text{definition test1 where}
\text{test1 = test-kil E list-name [Class list-name] Void 3 0}
\text{[Suc 0, 2, NullPointer, 7, 0]} \text{ append-ins}

\text{definition test2 where}
\text{test2 = test-kil E test-name [] Void 3 2 [] make-list-ins}

\text{definition test3 where test3 = } \varphi_a

\text{definition test4 where test4 = } \varphi_m

\text{ML-val \{ if } @\{\text{code test1}\} = @\{\text{code map}\} @\{\text{code OK}\} @\{\text{code test3}\} \text{ then } () \text{ else error wrong result;}
\text{ if } @\{\text{code test2}\} = @\{\text{code map}\} @\{\text{code OK}\} @\{\text{code test4}\} \text{ then } () \text{ else error wrong result }\}

\text{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
  | max-varsss :: '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(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-varsss 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-varsss [] = 0
  | max-varsss (e#es) = max (max-vars e) (max-varsss es)

  inductive
  | eval_1 :: J_1-prog ⇒ expr_1 ⇒ state_1 ⇒ expr_1 ⇒ state_1 ⇒ bool
    (⇒⇒ ((1(-,-)) ⇒/ (1(-,-))) [51,0,0,0,0] 81)
  | evals_1 :: J_1-prog ⇒ expr_1 list ⇒ state_1 ⇒ expr_1 list ⇒ state_1 ⇒ bool
    (⇒⇒ ((1(-,-)) ⇒/ (1(-,-))) [51,0,0,0,0] 81)
  for P :: J_1-prog
  where
New1:
[ new-Addr $h = \text{Some } a$; $P \vdash C \text{ has-fields FDTs}$; $h' = h(a \rightarrow (C, \text{init-fields FDTs}))$ ]

$\implies P \vdash_1 \langle \text{new } C,(h,l) \rangle \Rightarrow \langle \text{addr } a,(h',l) \rangle$

NewFail1:

new-Addr $h = \text{None} \implies P \vdash_1 \langle \text{new } C, (h,l) \rangle \Rightarrow \langle \text{THROW OutOfMemory}, (h,l) \rangle$

Cast:

[ $P \vdash_1 \langle e,s_0 \rangle \Rightarrow \langle \text{addr } a,(h,l) \rangle$; $h a = \text{Some}(D,fs)$; $P \vdash D \preceq^* C$ ]

$\implies P \vdash_1 \langle \text{Cast } C e,s_0 \rangle \Rightarrow \langle \text{addr } a,(h,l) \rangle$

CastNull1:

$P \vdash_1 \langle e,s_0 \rangle \Rightarrow \langle \text{null},s_1 \rangle \implies P \vdash_1 \langle \text{Cast } C e,s_0 \rangle \Rightarrow \langle \text{null},s_1 \rangle$

CastFail1:

[ $P \vdash_1 \langle e,s_0 \rangle \Rightarrow \langle \text{addr } a,(h,l) \rangle$; $h a = \text{Some}(D,fs)$; $\neg P \vdash D \preceq^* C$ ]

$\implies P \vdash_1 \langle \text{Cast } C e,s_0 \rangle \Rightarrow \langle \text{THROW ClassCast}, (h,l) \rangle$

CastThrow1:

$P \vdash_1 \langle e,s_0 \rangle \Rightarrow \langle \text{throw } e',s_1 \rangle \implies P \vdash_1 \langle \text{Cast } C e,s_0 \rangle \Rightarrow \langle \text{throw } e',s_1 \rangle$

Val1:

$P \vdash_1 \langle \text{Val } v,s \rangle \Rightarrow \langle \text{Val } v,s \rangle$

BinOp:

[ $P \vdash_1 \langle e_1,s_0 \rangle \Rightarrow \langle \text{Val } v_1,s_1 \rangle$; $P \vdash_1 \langle e_2,s_1 \rangle \Rightarrow \langle \text{Val } v_2,s_2 \rangle$; $\text{binop}(bop,v_1,v_2) = \text{Some } v$ ]

$\implies P \vdash_1 \langle e_1 <bop> e_2,s_0 \rangle \Rightarrow \langle \text{Val } v,s_2 \rangle$

BinOpThrow1:

$P \vdash_1 \langle e_1,s_0 \rangle \Rightarrow \langle \text{throw } e,s_1 \rangle \implies P \vdash_1 \langle e_1 <bop> e_2,s_0 \rangle \Rightarrow \langle \text{throw } e,s_1 \rangle$

BinOpThrow2:

[ $P \vdash_1 \langle e_1,s_0 \rangle \Rightarrow \langle \text{Val } v_1,s_1 \rangle$; $P \vdash_1 \langle e_2,s_1 \rangle \Rightarrow \langle \text{throw } e,s_2 \rangle$ ]

$\implies P \vdash_1 \langle e_1 <bop> e_2,s_0 \rangle \Rightarrow \langle \text{throw } e,s_2 \rangle$

Var1:

[ $ls[i = v; i < \text{size } ls]$ ]

$P \vdash_1 \langle \text{Var } i,(h,ls) \rangle \Rightarrow \langle \text{Val } v,(h,ls) \rangle$

LAss1:

[ $P \vdash_1 \langle e,s_0 \rangle \Rightarrow \langle \text{Val } v,(h,ls) \rangle$; $i < \text{size } ls$; $ls' = ls[i := v]$ ]

$\implies P \vdash_1 \langle i := e,s_0 \rangle \Rightarrow \langle \text{unit},(h,ls') \rangle$

LAssThrow1:

$P \vdash_1 \langle e,s_0 \rangle \Rightarrow \langle \text{throw } e',s_1 \rangle \implies P \vdash_1 \langle i := e,s_0 \rangle \Rightarrow \langle \text{throw } e',s_1 \rangle$

FAcc1:

[ $P \vdash_1 \langle e,s_0 \rangle \Rightarrow \langle \text{addr } a,(h,ls) \rangle$; $h a = \text{Some}(C,fs)$; $fs(F,D) = \text{Some } v$ ]

$\implies P \vdash_1 \langle e.F(D),s_0 \rangle \Rightarrow \langle \text{Val } v,(h,ls) \rangle$

FAccNull1:

$P \vdash_1 \langle e,s_0 \rangle \Rightarrow \langle \text{null},s_1 \rangle \implies P \vdash_1 \langle e.F(D),s_0 \rangle \Rightarrow \langle \text{THROW NullPointer},s_1 \rangle$

FAccThrow1:

$P \vdash_1 \langle e,s_0 \rangle \Rightarrow \langle \text{throw } e',s_1 \rangle \implies P \vdash_1 \langle e.F(D),s_0 \rangle \Rightarrow \langle \text{throw } e',s_1 \rangle$
| \text{FAss}_1: |
| \quad [P \vdash (\text{addr } a, s_1); P \vdash (\text{Val } v, (h_2, l_2))]; |
| \quad h_2 \; a = \text{Some}(C, fs); \; fs' = \text{fs}((F, D) \rightarrow v); \; h_2' = h_2(a \rightarrow (C, fs')) |
| \quad \Rightarrow \; P \vdash (e_1, F[D] := e_2, s_0) \Rightarrow \langle \text{unit}, (h_2', l_2) \rangle |
| \text{FAssNull}_1: |
| \quad [P \vdash (null, s_1); P \vdash (\text{Val } v, s_2)] |
| \quad \Rightarrow \; P \vdash (e_1, F[D] := e_2, s_0) \Rightarrow \langle \text{THROW NullPointer}, s_2 \rangle |
| \text{FAssThrow}_1: |
| \quad P \vdash (e_1, s_0) \Rightarrow \langle \text{throw } e', s_1 \rangle |
| \quad P \vdash (e_1, F[D] := e_2, s_0) \Rightarrow \langle \text{throw } e', s_1 \rangle |
| \text{CallObjThrow}_1: |
| \quad P \vdash (e_1, s_0) \Rightarrow \langle \text{throw } e', s_1 \rangle |
| \quad P \vdash (e_1, M(es), s_0) \Rightarrow \langle \text{throw } e', s_1 \rangle |
| \text{CallNull}_1: |
| \quad [P \vdash (e_1, s_0) \Rightarrow (null, s_1); P \vdash (es, s_1) \Rightarrow \langle \text{map } Val vs, s_2 \rangle] |
| \quad \Rightarrow \; P \vdash (e_1, M(es), s_0) \Rightarrow \langle \text{THROW NullPointer}, s_2 \rangle |
| \text{Call}: |
| \quad [P \vdash (e_1, s_0) \Rightarrow (addr a, s_1); P \vdash (es, s_1) \Rightarrow \langle \text{map } Val vs, (h_2, ls_2) \rangle]; |
| \quad h_2 \; a = \text{Some}(C, fs); \; P \vdash C \text{ sees } M: Ts \rightarrow T = \text{body in } D; |
| \quad \text{size } vs = \text{size } Ts; \; ls_2 = (\text{Addr } a) \# vs \# \text{ replicate (max-vars body) undefined}; |
| \quad P \vdash (\text{body}, (h_2, ls_2)) \Rightarrow \langle e', (h_3, ls_3) \rangle |
| \quad \Rightarrow \; P \vdash (e_1, M(es), s_0) \Rightarrow \langle e', (h_3, ls_2) \rangle |
| \text{CallParamsThrow}_1: |
| \quad [P \vdash (e_1, s_0) \Rightarrow (\text{Val } v, s_1); P \vdash (es, s_1) \Rightarrow \langle es', s_2 \rangle]; |
| \quad es' = \text{map } Val vs @ (\text{throw } ex @ \# es_2) |
| \quad \Rightarrow \; P \vdash (e_1, M(es), s_0) \Rightarrow \langle \text{throw } ex, s_2 \rangle |
| \text{Block}: |
| \quad P \vdash (e, s_0) \Rightarrow \langle e', s_1 \rangle \Rightarrow \; P \vdash (\text{Block } i \; T \; e, s_0) \Rightarrow \langle e', s_1 \rangle |
| \text{Seq}: |
| \quad [P \vdash (e_0, s_0) \Rightarrow (\text{Val } v, s_1); P \vdash (e_1, s_1) \Rightarrow (e_2, s_2)] |
| \quad \Rightarrow \; P \vdash (e_0; e_1, s_0) \Rightarrow (e_2, s_2) |
| \text{SeqThrow}_1: |
| \quad P \vdash (e_0, s_0) \Rightarrow \langle \text{throw } e, s_1 \rangle \Rightarrow \; P \vdash (e_0; e_1, s_0) \Rightarrow \langle \text{throw } e, s_1 \rangle |
| \text{CondT}_1: |
| \quad [P \vdash (e, s_0) \Rightarrow (\text{true}, s_1); P \vdash (e_1, s_1) \Rightarrow \langle e', s_2 \rangle] |
| \quad \Rightarrow \; P \vdash (\text{if } (e) \; e_1 \; \text{else } e_2, s_0) \Rightarrow \langle e', s_2 \rangle |
| \text{CondF}_1: |
| \quad [P \vdash (e, s_0) \Rightarrow (\text{false}, s_1); P \vdash (e_1, s_1) \Rightarrow \langle e', s_2 \rangle] |
| \quad \Rightarrow \; P \vdash (\text{if } (e) \; e_1 \; \text{else } e_2, s_0) \Rightarrow \langle e', s_2 \rangle |
| \text{CondThrow}_1: |
| \quad P \vdash (e, s_0) \Rightarrow \langle \text{throw } e', s_1 \rangle \Rightarrow \; P \vdash (\text{if } (e) \; e_1 \; \text{else } e_2, s_0) \Rightarrow \langle \text{throw } e', s_1 \rangle |
| \text{WhileF}_1: |
| \quad P \vdash (e, s_0) \Rightarrow \langle \text{false}, s_1 \rangle \Rightarrow \;
$$\begin{align*}
P & \vdash \{\text{while} \ (e) \ c, s_0\} \Rightarrow \{\text{unit}, s_1\} \\
\mid \text{While}T_1: & \quad \left[
\begin{align*}
P & \vdash \{e, s_0\} \Rightarrow \{\text{true}, s_1\}; \ P \vdash \{c, s_1\} \Rightarrow \{\text{Val} \ v_1, s_2\}; \\
P & \vdash \{\text{while} \ (e) \ c, s_2\} \Rightarrow \{e_3, s_3\}\end{align*}
\right] \\
& \Rightarrow \ P \vdash \{\text{while} \ (e) \ c, s_0\} \Rightarrow \{e_3, s_3\}
\end{align*}
$$

$$\begin{align*}
P & \vdash \{e, s_0\} \Rightarrow \{\text{true}, s_1\}; \ P \vdash \{c, s_1\} \Rightarrow \{\text{throw} \ e', s_2\}\end{align*}
$$

$$\begin{align*}
P & \vdash \{\text{while} \ (e) \ c, s_0\} \Rightarrow \{\text{throw} \ e', s_1\}
\end{align*}
$$

$$\begin{align*}
P & \vdash \{e, s_0\} \Rightarrow \{\text{throw} \ e', s_1\} \\
& \Rightarrow \ P \vdash \{\text{while} \ (e) \ c, s_0\} \Rightarrow \{\text{throw} \ e', s_2\}
\end{align*}
$$

$$\begin{align*}
P & \vdash \{e, s_0\} \Rightarrow \{\text{addr} \ a, s_1\} \\
& \Rightarrow \ P \vdash \{\text{throw} \ e, s_0\} \Rightarrow \{\text{Throw} \ a, s_1\}
\end{align*}
$$

$$\begin{align*}
P & \vdash \{e, s_0\} \Rightarrow \{\text{null}, s_1\} \\
& \Rightarrow \ P \vdash \{\text{throw} \ e, s_0\} \Rightarrow \{\text{THROW} \ \text{NullPointerException}, s_1\}
\end{align*}
$$

$$\begin{align*}
P & \vdash \{\text{throw} \ e, s_0\} \Rightarrow \{\text{throw} \ e', s_1\} \\
& \Rightarrow \ P \vdash \{\text{throw} \ e, s_0\} \Rightarrow \{\text{throw} \ e', s_1\}
\end{align*}
$$

$$\begin{align*}
P & \vdash \{e_1, s_0\} \Rightarrow \{\text{Val} \ v_1, s_1\} \\
& \Rightarrow \ P \vdash \{\text{try} \ e_1 \ \text{catch} (C \ i) \ e_2, s_0\} \Rightarrow \{\text{Val} \ v_1, s_1\}
\end{align*}
$$

$$\begin{align*}
P & \vdash \{e_1, s_0\} \Rightarrow \{\text{Throw} \ a, (h_1, l_1)\} \\
& \Rightarrow \ P \vdash \{\text{try} \ e_1 \ \text{catch} (C \ i) \ e_2, s_0\} \Rightarrow \{\text{Throw} \ a, (h_1, l_1)\}
\end{align*}
$$

$$\begin{align*}
P & \vdash \{e_1, s_0\} \Rightarrow \{\text{throw} \ e', s_1\} \\
& \Rightarrow \ P \vdash \{e_1, s_0\} \Rightarrow \{\text{throw} \ e' \ # \ es, s_1\}
\end{align*}
$$

$$\begin{align*}
P & \vdash \{e_0, (h_0, l_0)\} \Rightarrow \{e_1, (h_1, l_1)\} \Rightarrow \text{length} \ l_0 = \text{length} \ l_1
\end{align*}
$$

$$\begin{align*}
P & \vdash \{e_0, (h_0, l_0)\} \Rightarrow \{e_1, (h_1, l_1)\} \Rightarrow \text{length} \ l_0 = \text{length} \ l_1
\end{align*}
$$

$$\begin{align*}
\text{lemma eval1-preserves-len:} \\
P & \vdash \{e_0, (h_0, l_0)\} \Rightarrow \{e_1, (h_1, l_1)\} \Rightarrow \{e_1, (h_1, l_1)\} \Rightarrow \text{length} \ l_0 = \text{length} \ l_1
\end{align*}
$$

$$\begin{align*}
\text{lemma eval1-preserves-len:} \\
P & \vdash \{e_0, (h_0, l_0)\} \Rightarrow \{e_1, (h_1, l_1)\} \Rightarrow \{e_1, (h_1, l_1)\} \Rightarrow \text{length} \ l_0 = \text{length} \ l_1
\end{align*}
$$

$$\begin{align*}
\forall \ es' \ s'. \ P & \vdash \{es, s\} \Rightarrow \{es', s'\} \Rightarrow \text{length} \ es = \text{length} \ es'
\end{align*}
$$

$$\begin{align*}
\text{lemma eval1-final:} \ P & \vdash \{e, s\} \Rightarrow \{e', s'\} \Rightarrow \text{final} \ e' \\
\text{and eval1-final:} \ P & \vdash \{es, s\} \Rightarrow \{es', s'\} \Rightarrow \text{finals} \ es'
\end{align*}
$$
end

5.2 Well-Formedness of Intermediate Language

theory J1WellForm
imports ../J/JWellForm J1
begin

5.2.1 Well-Typedness

type-synonym
env1 = ty list — type environment indexed by variable number

inductive
WT1 :: [J1-prog, env1, expr1, ty] ⇒ bool
((\_, \_ :: \_) [51, 51, 51])

and WTS1 :: [J1-prog, env1, expr1 list, ty list] ⇒ bool
((\_, \_ :: [\_]) [51, 51, 51])

for P :: J1-prog

where

WTNew1:
is-class P C ⇒ P, E ⊢ new C :: Class C

| WTCast1:
\[ P, E ⊢ e :: Class D; \ is-class P C; \ P ⊢ C \leq D \lor P ⊢ D \leq C \]\n⇒ P, E ⊢ Cast C e :: Class C

| WTVal1:
typeof v = Some T ⇒
P, E ⊢ Val v :: T

| WTVar1:
\[ E!i = T; \ i < size E \]\n⇒ P, E ⊢ Var i :: T

| WTBinOp1:
\[ P, E ⊢ e1 :: T1; \ P, E ⊢ e2 :: T2;\]
\[ \ case \ bop \ of \ Eq \ ⇒ (P ⊢ T1 \leq T2 \lor P ⊢ T2 \leq T1) \land T = Boolean \]
| Add ⇒ T1 = Integer \land T2 = Integer \land T = Integer \]
⇒ P, E ⊢ e1 \ldt{bop} e2 :: T

| WTLAss1:
\[ E!i = T; \ i < size E; P, E ⊢ e :: T'; P ⊢ T' \leq T \]\n⇒ P, E ⊢ i:=e :: Void

| WTAcc1:
\[ P, E ⊢ e :: Class C; \ P ⊢ C sees F:T in D \]\n⇒ P, E ⊢ e·F\{D\} :: T

| WTAss1:
\[ P, E ⊢ e1 :: Class C; \ P ⊢ C sees F:T in D; P, E ⊢ e2 :: T'; P ⊢ T' \leq T \]
\[ P, E \vdash e_1 \cdot F\{D\} ::= e_2 :: \text{Void} \]

| **WTCall** | \[ P, E \vdash e :: \text{Class C}; P \vdash C \text{ sees M} : Ts' \rightarrow T = m \text{ in } D; \]  
| | \[ P, E \vdash \text{es} :: Ts; P \vdash Ts [\leq] Ts' \]  
| | \[ P, E \vdash e \cdot M(\text{es}) :: T \]  

| **WTBlock** | \[ \text{is-type } P T; P, E@[T] \vdash e :: T' \]  
| | \[ P, E \vdash \{ i : T; e \} :: T' \]  

| **WTSeq** | \[ P, E \vdash e_1 :: T_1; P, E \vdash e_2 :: T_2 \]  
| | \[ P, E \vdash e_1 ; e_2 :: T_2 \]  

| **WTCond** | \[ P, E \vdash e :: \text{Boolean}; P, E \vdash e_1 :: T_1; P, E \vdash e_2 :: T_2; \]  
| | \[ P \vdash T_1 \leq T_2 \lor P \vdash T_2 \leq T_1; P \vdash T_1 \leq T_2 \Rightarrow T = T_2; P \vdash T_2 \leq T_1 \Rightarrow T = T_1 \]  
| | \[ P, E \vdash \text{if } (e) e_1 \text{ else } e_2 :: T \]  

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

| **WTTry** | \[ P, E \vdash e :: \text{Class C} \Rightarrow \]  
| | \[ P, E \vdash \text{throw } e :: \text{Void} \]  

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

| **WTCons** | \[ P, E \vdash e :: T; P, E \vdash \text{es} :: Ts \]  
| | \[ P, E \vdash e \# \text{es} :: T \# Ts \]  

**lemma** WT1-same-size: \[\forall Ts. P, E \vdash \text{es} :: Ts \Rightarrow \text{size es} = \text{size Ts}\]  

**lemma** WT1-unique:  
\[ P, E \vdash e :: T_1 \Rightarrow (\forall T_2. P, E \vdash e :: T_2 \Rightarrow T_1 = T_2) \text{ and}  
\]  
\[ P, E \vdash \text{es} :: Ts_1 \Rightarrow (\forall Ts_2. P, E \vdash \text{es} :: Ts_2 \Rightarrow Ts_1 = Ts_2) \]  

**lemma assumes** \( w t : \text{wf-prog } p P \)  
**shows** WT1-is-type: \[ P, E \vdash e :: T \Rightarrow \text{set } E \subseteq \text{types } P \Rightarrow \text{is-type } P T \]  
and \[ P, E \vdash \text{es} :: Ts \Rightarrow \text{True} \]  

### 5.2.2 Well-formedness

— Indices in blocks increase by 1

**primrec** \( \text{B} :: \text{expr} \rightarrow \text{nat} \rightarrow \text{bool} \)
and $B$s :: $expr_1$ list $\Rightarrow$ nat $\Rightarrow$ bool where

$B$ (new $C$) $i$ = $True$ $\mid$
$B$ (Cast $C$ $e$) $i$ = $B$ $e$ $i$ $\mid$
$B$ (Val $v$) $i$ = $True$ $\mid$
$B$ ($e_1$ <bop> $e_2$) $i$ = ($B$ $e_1$ $i$ $\land$ $B$ $e_2$ $i$) $\mid$
$B$ (Var $j$) $i$ = $True$ $\mid$
$B$ ($e$-$F$($D$)) $i$ = $B$ $e$ $i$ $\mid$
$B$ ($j$ ::= $e$) $i$ = $B$ $e$ $i$ $\mid$
$B$ ($e_1$-$F$($D$) ::= $e_2$) $i$ = ($B$ $e_1$ $i$ $\land$ $B$ $e_2$ $i$) $\mid$
$B$ ($e$-$M$(es)) $i$ = ($B$ $e$ $i$ $\land$ $B$s es $i$) $\mid$
$B$ ($\{j; T ; e\}$) $i$ = ($i$ = $j$ $\land$ $B$ $e$ ($i$+1)) $\mid$
$B$ ($e_1$::$e_2$) $i$ = ($B$ $e_1$ $i$ $\land$ $B$ $e_2$ $i$) $\mid$
$B$ ($if$ ($e$) $e_1$ else $e_2$) $i$ = ($B$ $e$ $i$ $\land$ $B$ $e_1$ $i$ $\land$ $B$ $e_2$ $i$) $\mid$
$B$ (throw $e$) $i$ = $B$ $e$ $i$ $\mid$
$B$ (while ($e$) $c$) $i$ = ($B$ $e$ $i$ $\land$ $B$ $c$ $i$) $\mid$
$B$ (try $e_1$ catch($C$) $j$) $e_2$) $i$ = ($B$ $e_1$ $i$ $\land$ $i$ = $j$ $\land$ $B$ $e_2$ ($i$+1)) $\mid$

$B$s [[] $i$ = $True$ $\mid$
$B$s ($e$#es) $i$ = ($B$ $e$ $i$ $\land$ $B$s es $i$)

definition $wf$-$J_1$-$mdecl$ :: $J_1$-prog $\Rightarrow$ cname $\Rightarrow$ $expr_1$ mdecl $\Rightarrow$ bool where

$wf$-$J_1$-$mdecl$ $P$ $C$ $\equiv$ $\lambda$(M,Ts,T,body).
($\exists$ $T'$, $P$,Class $C$#Ts $\vdash_1$ body :: $T'$ $\land$ $P$ $\vdash$ $T'$ $\leq$ $T$) $\land$
$D$ body [{$..$ size Ts$]} $\land$ $B$ body (size Ts $+ 1$)

lemma $wf$-$J_1$-$mdecl$[simp]:
$wf$-$J_1$-$mdecl$ $P$ $C$ (M,Ts,T,body) $\equiv$
($\exists$ $T'$, $P$,Class $C$#Ts $\vdash_1$ body :: $T'$ $\land$ $P$ $\vdash$ $T'$ $\leq$ $T$) $\land$
$D$ body [{$..$ size Ts$]} $\land$ $B$ body (size Ts $+ 1$)

abbreviation $wf$-$J_1$-prog $\equiv$ $wf$-prog $wf$-$J_1$-$mdecl$

end

5.3 Program Compilation

thory PCompiler
imports ../Common/WellForm
begin

definition $comp$-$M$ :: ($'a$ $\Rightarrow$ $'b$) $\Rightarrow$ $'a$ mdecl $\Rightarrow$ $'b$ mdecl where

$comp$-$M$ $f$ $\equiv$ $\lambda$(M, Ts, T, m). (M, Ts, T, f m)

definition $comp$-$C$ :: ($'a$ $\Rightarrow$ $'b$) $\Rightarrow$ $'a$ cdecl $\Rightarrow$ $'b$ cdecl where

$comp$-$C$ $f$ $\equiv$ $\lambda$(C,D,Fdecls,Mdecls). (C,D,Fdecls, map ($comp$-$M$ $f$) Mdecls)

definition $comp$-$P$ :: ($'a$ $\Rightarrow$ $'b$) $\Rightarrow$ $'a$ prog $\Rightarrow$ $'b$ prog where

$comp$-$P$ $f$ $\equiv$ map ($comp$-$C$ $f$)

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`:

\[ \text{map-of} \ (\text{map} \ (\lambda(x,a,b,c).(x,a,b,f \ c)) \ ts) = \]

\[ \text{map-option} \ (\lambda(a,b,c).(a,b,f \ c)) \circ (\text{map-of} \ ts) \]

**lemma** `class-compP`:

\[ \text{class} \ P \ C = \text{Some} \ (D, \ fs, \ ms) \]

\[ \implies \ \text{class} \ (\text{compP} \ f \ P) \ C = \text{Some} \ (D, \ fs, \map \ (\text{compM} \ f) \ ms) \]

**lemma** `class-compPD`:

\[ \text{class} \ (\text{compP} \ f \ P) \ C = \text{Some} \ (D, \ fs, \ ms) \]

\[ \implies \ \exists \ ms. \ \text{class} \ P \ C = \text{Some} \ (D, \ fs, \ms) \land \ \text{ms} = \text{map} \ (\text{compM} \ f) \ ms \]

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

**lemma** `[simp]` `class (\text{compP} \ f \ P) \ C = \text{map-option} \ (\lambda(a,b,c).((f \ c),D)) \circ (\text{class} \ P \ C)`

**lemma** `sees-methods-compP`:

\[ P \vdash C \ \text{sees-methods} \ Mm \]

\[ \implies \ compP \ f \ P \vdash C \ \text{sees-methods} \ (\text{map-option} \ (\lambda((Ts,T,m),D). ((Ts,T,f \ m),D)) \circ Mm) \]

**lemma** `sees-method-compP`:

\[ P \vdash C \ \text{sees} \ M : Ts \rightarrow T = m \ in \ D \]

\[ \implies \ compP \ f \ P \vdash C \ \text{sees} \ M : Ts \rightarrow T = (f \ m) \ in \ D \]

**lemma** `[simp]` `method (\text{compP} \ f \ P) \ C \ M = (D, Ts, T, f m)`

**lemma** `sees-methods-compPD`:

\[ \exists \ Mm. \ P \vdash C \ \text{sees-methods} \ Mm \land \]

\[ Mm' = (\text{map-option} \ (\lambda((Ts,T,m),D). ((Ts,T,f \ m),D)) \circ Mm) \]

**lemma** `sees-method-compPD`:

\[ \exists \ m. \ P \vdash C \ \text{sees} \ M : Ts \rightarrow T = m \ in \ D \land f \ m = fm \]

**lemma** `[simp]` `subcls1 (\text{compP} \ f \ P) = subcls1 \ P`

**lemma** `compP-widen` `[simp]` `(\text{compP} \ f \ P \vdash T \leq T') = (P \vdash T \leq T')`

**lemma** `[simp]` `(\text{compP} \ f \ P \vdash Ts \leq T') = (P \vdash Ts \leq T')`

**lemma** `[simp]` `is-type (\text{compP} \ f \ P) \ T = is-type \ P \ T`

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

**lemma** `[simp]` `fields (\text{compP} \ f \ P) \ C = fields \ P \ C`

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

**lemma** `[simp]` `field (\text{compP} \ f \ P) \ F \ D = field \ P \ F \ D`
5.3.1 Invariance of \texttt{wf-prog} under compilation

\textbf{lemma} \[\text{iff}]: \ \text{distinct-fst} \ (\text{compP} f P) = \text{distinct-fst} P

\textbf{lemma} \[\text{iff}]: \ \text{distinct-fst} \ (\text{map} \ (\text{compM} f) \ ms) = \text{distinct-fst} ms

\textbf{lemma} \[\text{iff}]: \ \text{wf-syscls} \ (\text{compP} f P) = \text{wf-syscls} P

\textbf{lemma} \[\text{iff}]: \ \text{wf-fdecl} \ (\text{compP} f P) = \text{wf-fdecl} P

\textbf{lemma} \ \text{set-compP}:

\[(\exists \ ms. \ (C,D,fs,ms') \in \text{set}(\text{compP} f P)) =\]
\[
\exists ms. \ (C,D,fs,ms) \in \text{set} P \land ms' = \text{map} \ (\text{compM} f) \ ms\]

\textbf{lemma} \ \text{wf-cdecl-compPI}:

\[(\forall C M Ts T m. \ [\ [ \text{wf-mdecl} \ \text{wf} 1 \ P C (M,Ts,T,m); \ P \vdash \ C \ \text{sees} \ M:Ts \rightarrow T = m \ \text{in} \ C \ ]\]
\[
\Rightarrow \text{wf-cdecl} \ \text{wf} 2 \ (\text{compP} f P) \ C (M,Ts,T,f m);\]
\[
\forall x \in \text{set} P. \ \text{wf-cdecl} \ \text{wf} 1 \ P x; \ x \in \text{set} (\text{compP} f P); \ \text{wf-prog} p P \ ]\]
\[
\Rightarrow \text{wf-cdecl} \ \text{wf} 2 \ (\text{compP} f P) x\]

\textbf{lemma} \ \text{wf-prog-compPI}:

\textbf{assumes lift}:

\[(\forall C M Ts T m. \ [P \vdash \ C \ \text{sees} \ M:Ts \rightarrow T = m \ \text{in} \ C; \ \text{wf-mdecl} \ \text{wf} 1 \ P C (M,Ts,T,m) ]\]
\[
\Rightarrow \text{wf-mdecl} \ \text{wf} 2 \ (\text{compP} f P) C (M,Ts,T,f m)\]

\textbf{and} \ \text{wf} : \ \text{wf-prog} \ \text{wf} 1 \ P \]

\textbf{shows} \ \text{wf-prog} \ \text{wf} 2 \ (\text{compP} f P)

end

\textit{theory} \ \texttt{List-Index} \ \textit{imports} \ \texttt{Main} \ \begin{flushleft}
\text{This theory collects functions for index-based manipulation of lists.}
\end{flushleft}

5.3.2 Finding an index

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

\textit{find-index} \ \texttt{P xs} \ finds the index of the first element in \texttt{xs} that satisfies \texttt{P}.

\textit{index} \ \texttt{xs x} \ finds the index of the first occurrence of \texttt{x} in \texttt{xs}.

\textit{last-index} \ \texttt{xs x} \ finds the index of the last occurrence of \texttt{x} in \texttt{xs}.

All functions return \textit{length} \texttt{xs} if \texttt{xs} does not contain a suitable element.

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

\textbf{primrec} \ \textit{find-index} :: \ (\texttt{'a \Rightarrow bool}) \Rightarrow \ 'a \ \texttt{list} \Rightarrow \ \texttt{nat} \ \textbf{where}

\textit{find-index} - \ [ ] = \ 0 \ |

\textit{find-index} \ \texttt{P (x\#xs)} = \ (\textit{if} \ \texttt{P} \ \texttt{x} \ \texttt{then} \ 0 \ \texttt{else} \ \textit{find-index} \ \texttt{P} \ \texttt{xs + 1})
definition index :: 'a list ⇒ 'a ⇒ nat where
index xs = (λa. find-index (λx. x=a) xs)

definition last-index :: 'a list ⇒ 'a ⇒ 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 ∉ set xs ⇒ index xs x = size xs
by (induct xs) auto

lemma find-index-eq-size-conv:
  size xs = n ⇒ (find-index P xs = n) = (ALL x : set xs. ¬ P x)
by (induct xs arbitrary: n) auto

lemma size-eq-find-index-conv:
  size xs = n ⇒ (n = find-index P xs) = (ALL x : set xs. ¬ P x)
by (metis find-index-eq-size-conv)

lemma index-size-conv: size xs = n ⇒ (index xs x = n) = (x ∉ 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:
\[(\text{find-index } P \, \text{xs} < \text{size xs}) = (\text{EX } x : \text{set xs}. \ P \, x)\]
by (induct xs) auto

lemma index-less-size-conv:
\[(\text{index xs } x < \text{size xs}) = (x \in \text{set xs})\]
by (auto simp: index-def find-index-less-size-conv)

lemma last-index-less-size-conv:
\[(\text{last-index xs } x < \text{size xs}) = (x \in \text{set xs})\]
by (simp add: last-index-less-size-conv simp_all)

lemma index-less[simp]:
\[x : \text{set xs} \Rightarrow \text{size xs} \leq n \Rightarrow \text{index xs } x < n\]
apply (induct xs) apply auto
apply (metis index-less-size-conv less-eq-Suc-le less-trans-Suc)
done

lemma last-index-less[simp]:
\[x : \text{set xs} \Rightarrow \text{size xs} \leq n \Rightarrow \text{last-index xs } x < n\]
by (simp add: last-index-less-size-conv symmetric)

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

lemma last-index-append: last-index \((\text{xs @ ys})\) \(x\) =
(if \(x : \text{set ys}\) then size \(\text{xs}\) + last-index \(\text{ys}\) \(x\)
  else if \(x : \text{set xs}\) then last-index \(\text{xs}\) \(x\) else size \(\text{xs}\) + size \(\text{ys}\))
by (induct xs) (simp_all add: last-index-Cons last-index-size-conv)

lemma last-index-Snoc[simp]:
last-index \((\text{xs @ } [x])\) \(y\) =
(if \(x = y\) then size \(\text{xs}\)
  else if \(y : \text{set xs}\) then last-index \(\text{xs}\) \(y\) else size \(\text{xs}\) + 1)
by (simp add: last-index-append last-index-Cons)

lemma nth-find-index: find-index \(P \, \text{xs} < \text{size xs}\) \(\Rightarrow\) \(P(\text{xs} ! \text{find-index } P \, \text{xs})\)
by (induct xs) auto

lemma nth-index[simp]: \(x : \text{set xs} \Rightarrow \text{xs} ! \text{index xs } x = x\)
by (induct xs) auto

lemma nth-last-index[simp]: \(x : \text{set xs} \Rightarrow \text{xs} ! \text{last-index xs } x = x\)
by (simp add: last-index-def index-size-conv Let-def rev-nth symmetric)

lemma index-nth-id:
distinct xs; \ n < \ length \ xs \ \implies \ index \ xs \ (xs ! n) = n

by (metis in-set-conv-nth index-less-size-conv nth-eq-iff-index-eq nth-index)

lemma index-upl[simp]; \ m \leq i \implies i < n \implies index \ [m..<n] \ i = i - m

by (induction \ n) (auto simp add: index-append)

lemma index-eq-index-conv[simp]; \ x \in \ set \ xs \ \lor \ y \in \ set \ xs = \ \implies \ (index \ xs \ x = index \ xs \ y) = (x = y)

by (induct \ xs) auto

lemma inj-on-index: inj-on (index \ xs) (set \ xs)

by (simp add: inj-on-def)

lemma inj-on-index2: \ I \subseteq \ set \ xs \ \implies \ inj-on \ (index \ xs) \ I

by (rule inj-onI) auto

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 \ (\lambda \ y. x \neq y) \ xs)

by (induct \ xs) auto

lemma index-take: \ index \ xs \ x \geq i \implies x \notin \ set (take \ i \ xs)

apply (subgoal-tac \ set (take \ i \ xs) \subseteq \ set (takeWhile \ (op \neq \ 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 \implies x \notin \ 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: \ if-splits)

apply (metis \ rev-drop \ set-rev)

done

lemma set-take-if-index: \ assumes \ index \ xs \ x < i \ \land \ i \leq \ length \ xs

shows \ x \in \ set \ (take \ i \ xs)

proof

have \ index \ (take \ i \ xs @ \ drop \ i \ xs) \ x < i

using \ append-take-drop-id[of \ i \ xs] \ assms(1) \ by \ simp

thus \ \?thesis \ using \ assms(2)

by (simp \ add: \ index-append \ del: \ append-take-drop-id \ split: \ if-splits)

qed

lemma index-take-if-index:

assumes \ index \ xs \ x \leq \ n \ \implies \ index \ (take \ n \ xs) \ x = index \ xs \ x

proof cases

assume \ x : \ set \ (take \ n \ xs) \ with \ assms \ show \ \?thesis

by (metis \ append-take-drop-id \ index-append)
next
  assume \( x \notin \text{set}(\text{take} \ n \ \text{xs}) \) with \textbf{assms} show \( \text{?thesis} \)
  by (metis \text{order-le-less} \text{set-take-if-index} \text{le-cases} \text{length-take} \text{min-def} \text{size-index-conv} \text{take-all})
qed

lemma \text{index-take-if-set}:
  \( x : \text{set}(\text{take} \ n \ \text{xs}) \) \( \implies \) \( \text{index} \ \text{take} \ n \ \text{xs} \ x = \text{index} \ \text{xs} \ x \)
by (metis \text{index-take} \text{index-take-if-index} \text{linear})

lemma \text{index-last}:[simp]:
  \( \text{xs} \neq [] \implies \text{distinct} \ \text{xs} \implies \text{index} \ \text{xs} \ (\text{last} \ \text{xs}) = \text{length} \ \text{xs} - 1 \)
by (induction \text{xs}) auto

lemma \text{index-update-if-distinct}: \( n < \text{length} \ \text{xs} \implies x \neq \text{xs}[n] \implies x \neq y \implies \text{index} \ (\text{xs}[n := y]) \ = \text{index} \ \text{xs} \ x \)
by (subst (2) \text{id-take-nth-drop[of n \ \text{xs}})
(auto simp: \text{upd-conv-take-nth-drop} \text{index-append} \text{min-def})

lemma \text{(set-drop-if-index}):
  \( \text{distinct} \ \text{xs} \implies \text{index} \ \text{xs} \ x < i \implies x \notin \text{set} \ \text{drop} \ i \ \text{xs} \)
by (metis \text{in-set-dropD} \text{index-nth-id} \text{last-index-drop} \text{last-index-less-size-conv} \text{nth-last-index})

lemma \text{list-update-swap}:
  \( n < \text{length} \ \text{xs} \implies x \neq \text{xs}[n] \implies x \neq y \implies x \neq \text{ys}[j] \implies \text{index} \ \text{xs}[n := y, j := \text{ys}[j]] \ x = \)
(auto simp: \text{swap-def} simp del: \text{distinct-swap})
(auto simp: \text{index-nth-id} \text{length-list-update} \text{list-update-swap} \text{nth-list-update-eq})
(auto simp: \text{index-nth-id} \text{length-list-update} \text{nth-list-update-eq})
by (metis \text{index-update-if-distinct} \text{length-list-update} \text{nth-list-update})
qed

lemma \text{bij-betw-index}:
  \( \text{distinct} \ \text{xs} \implies X = \text{set} \ \text{xs} \implies l = \text{size} \ \text{xs} \implies \text{bij-betw} \ (\text{index} \ \text{xs}) X \{0..<l\} \)
apply simp
apply (rule \text{bij-betw-imageI[OF \text{inj-on-index}])
by (auto simp: \text{image-def} \text{bij-betw-imageE} \text{bij-betw-index})

5.3.3 Map with index

primrec \text{map-index}': nat \Rightarrow (nat \Rightarrow 'a \Rightarrow 'b) \Rightarrow 'a \text{ list} \Rightarrow 'b \text{ list} where
\text{map-index}' \ n \ f \ [] = []
| \text{map-index}' \ n \ f \ (x#xs) = f n \ x # \text{map-index}' (\text{Suc} \ n) \ f \ xs

lemma \text{length-map-index}':
  \text{length} \ (\text{map-index}' \ n \ f \ \text{xs}) = \text{length} \ \text{xs}
by (induct \text{xs} \ arbitrary: \ n) auto

lemma \text{map-index}'-\text{map-zip}:
  \text{map-index}' \ n \ f \ \text{xs} = \text{map} \ \text{split} \ f \ (\text{zip} \ [n..<n + \text{length} \ \text{xs}] \ \text{xs})
proof (induct \text{xs} \ arbitrary: \ n)
case (Cons x xs)
hence map-index' n f (x#xs) = f n x # map (split f) (zip [Suc n ..< n + length (x # xs)] xs) by simp
also have ... = map (split f) (zip (n # [Suc n ..< n + length (x # xs)]) (x # xs)) by simp
also have (n # [Suc n ..< n + length (x # xs)]) = [n ..< n + length (x # xs)] by (induct xs) auto
finally show ?case by simp
qed simp

abbreviation map-index ≡ map-index' 0

lemmas map-index = map-index'-map-zip[of 0, simplified]

lemma take-map-index: take p (map-index f xs) = map-index f (take p xs)
  unfolding map-index by (auto simp: min_def take-map take-zip)

lemma drop-map-index: drop p (map-index f xs) = map-index' p f (drop p xs)
  unfolding map-index'-map-zip by (cases p < length xs) (auto simp: drop-map drop-zip)

lemma map-map-index[simp]: map g (map-index f xs) = map-index (λn x. g (f n x)) xs
  unfolding map-index by auto

lemma map-index-map[simp]: map f (map g xs) = map-index (λn x. f n (g x)) xs
  unfolding map-index by (auto simp: map-zip-map2)

lemma set-map-index[simp]: x ∈ set (map-index f xs) = (∃i < length xs. f i (xs ! i) = x)
  unfolding map-index by (auto simp: set-zip intro!: image-eqI[of - split f])

lemma set-map-index'[simp]: x ∈ set (map-index' n f xs)
  <-> (∃i < length xs. f (n+i) (xs!i) = x)
  unfolding map-index'-map-zip
  by (auto simp: set-zip intro!: image-eqI[of - split f])

lemma nth-map-index[simp]: p < length xs ==> map-index f xs ! p = f p (xs ! p)
  unfolding map-index by auto

lemma map-index-cong:
  ∀p < length xs. f p (xs ! p) = g p (xs ! p) ==> map-index f xs = map-index g xs
  unfolding map-index by (auto simp: set-zip)

lemma map-index-id: map-index (curry snd) xs = xs
  unfolding map-index by auto

lemma map-index-no-index[simp]: map-index (λn x. f x) xs = map f xs
  unfolding map-index by (induct xs rule: rev-induct) auto

lemma map-index-congL:
  ∀p < length xs. f p (xs ! p) = xs ! p ==> map-index f xs = xs
  by (rule trans[OF map-index-cong map-index-id]) auto

lemma map-index'-is-NilD: map-index' n f xs = [] ==> xs = []
  by (induct xs) auto

declare map-index'-is-NilD[of 0, dest!]
theory List-Index

lemma map-index'-is-ConsD:
  map-index' \ n \ f \ xs = y \ # \ ys \implies \exists \ z \ zs. \ xs = z \ # \ zs \land f \ n \ z = y \land map-index' (n + 1) f zs = ys
by (induct xs arbitrary: n) auto

lemma map-index'-eq-imp-length-eq: map-index' \ n \ f \ xs = map-index' \ n \ g \ ys \implies length xs = length ys
proof (induct ys arbitrary: xs n)
  case (Cons y ys) thus \ ?case by (cases xs) auto
qed (auto dest: map-index'-is-NilD)

lemmas map-index-eq-imp-length-eq = map-index'-eq-imp-length-eq[of 0]

lemma map-index'-comp[simp]: map-index' \ n \ f \ (map-index' \ n \ g \ xs) = map-index' \ n \ (\lambda n. f \ n \ o \ g \ n) \ xs
by (induct xs arbitrary: n) auto

lemma map-index'-append[simp]: map-index' \ n \ f \ (a @ b) = map-index' \ n \ f \ a @ map-index' (n + length a) f b
by (induct a arbitrary: n) auto

lemma map-index-append[simp]: map-index f (a @ b) = map-index f a @ map-index' (length a) f b
using map-index-append[where n=0]
by (simp del: map-index'-append)

subsection{Insert at position}
primrec insert-nth :: nat \Rightarrow 'a \Rightarrow 'a list \Rightarrow 'a list
  | insert-nth (Suc n) x xs = (case xs of \ [] \Rightarrow [x] \ | y # ys \Rightarrow y # insert-nth n x ys)

lemma insert-nth-take-drop[simp]: insert-nth n x xs = take n xs @ [x] @ drop n xs
proof (induct n arbitrary: xs)
  case Suc thus \ ?case by (cases xs) auto
qed simp

lemma length-insert-nth: length (insert-nth n x xs) = Suc (length xs)
by (induct xs) auto

  Insert several elements at given (ascending) positions

lemma length-fold-insert-nth:
  length (fold (\lambda (p, b). insert-nth p b) pxs xs) = length xs + length pxs
by (induct pxs arbitrary: xs) auto

lemma invar-fold-insert-nth:
  \[ \forall x \in \set pxs. \ p < \fst x; \ p < \length xs; \ xs ! p = b \] \implies
  fold (\lambda (x, y). insert-nth x y) pxs xs ! p = b
by (induct pxs arbitrary: xs) (auto simp: nth-append)

lemma nth-fold-insert-nth:
  \[ \forall (p, b) \in \set pxs. \ p < \length xs + \length pxs; \ i < \length pxs; \ pxs ! i = (p, b) \] \implies
  fold (\lambda (p, b). insert-nth p b) pxs xs ! p = b
proof (induct pxs arbitrary: xs i p b)
case (Cons pb pxs)
show ?case
proof (cases i)
case 0
with Cons.prems have p < Suc (length xs)
proof (induct pxs rule: rev-induct)
case (snoc pb' pxs)
then obtain p' b' where pb' = (p', b') by auto
with snoc.prems have \( \forall p \in \text{fst } \text{set } \text{pxs}. \ p < p' \ p' \leq \text{Suc } (\text{length } \text{xs} + \text{length } \text{pxs}) \)
by (auto simp: image-iff sorted-Cons sorted-append le-eq-less-or-eq)
with snoc.prems show ?case by (intro snoc(1)) (auto simp: sorted-Cons sorted-append)
qed auto
with 0 Cons.prems show ?thesis unfolding fold.simps o-apply
by (intro invar-fold-insert-nth) (auto simp: sorted-Cons image-iff le-eq-less-or-eq nth-append)
next
case (Suc n) with Cons.prems show ?thesis unfolding fold.simps
by (auto intro: Cons(1) simp: sorted-Cons)
qed
qed simp

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: \( x \in \text{set } \text{xs} \implies \text{hidden } (\text{xs } @ [x]) \) (last-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 \implies \text{last-index } \text{xs} \ x \neq i
by (auto simp add: hidden-def last-index-drop last-index-less-size-conv)

lemma [simp]: hidden xs i \implies \text{hidden } (\text{xs}@[x]) \ i
by (auto simp add: hidden-def nth-append)

lemma fun-upds-apply:
(m(xs[→]ys)) x =
(let xs' = take (size ys) xs
in if x \in \text{set } \text{xs'} then Some(ys ! last-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:
\[(m(xs\mapsto ys)) \cdot x = \text{Some } y\] =
(\text{let } x' = \text{take (size } ys) \cdot xs \
\text{in if } x \in \text{set } x' \text{ then } ys ![last-index ] x' \cdot x = y \text{ else } m \cdot x = \text{Some } y)\]
by (simp add: fun-upds-apply Let-def)

lemma map-upds-upd-conv-last-index:
\[\forall x \in \text{set } xs; \text{ size } xs \leq \text{size } ys \implies m(xs\mapsto ys)\cdot(x\mapsto y) = m(xs\mapsto[y][\text{last-index } xs \cdot x := y])\]
apply (rule ext)
apply (simp add: fun-upds-apply eq-sym-conv Let-def)
done

5.4 Compilation Stage 1

theory Compiler1 imports PCompiler J1 Hidden begin

Replacing variable names by indices.

primrec compE1 :: vname list ⇒ expr ⇒ expr1 
and compE \_1 :: vname list ⇒ expr list ⇒ expr1 list where
compE1 Vs (new C) = new C  
| compE1 Vs (Cast C e) = Cast C (compE1 Vs e)  
| compE1 Vs (Var v) = Val v  
| compE1 Vs (e₁ ≲ bop ≲ e₂) = (compE1 Vs e₁) ≲ bop ≲ (compE1 Vs e₂)  
| compE1 Vs (Var V) = Var(last-index Vs V)  
| compE1 Vs (V := e) = (last-index Vs V) := (compE1 Vs e)  
| compE1 Vs (ecdot D) = (compE1 Vs e)cdot D  
| compE1 Vs (ecdot M(es)) = (compE1 Vs e)cdot M(compE₁ Vs es)  
| compE1 Vs \{V:T; e\} = \{(size Vs);T; compE1 (Vs@[V]) e\}  
| compE1 Vs (e₁;e₂) = (compE1 Vs e₁);(compE1 Vs e₂)  
| compE1 Vs (if (e) e₁ else e₂) = (if (compE₁ Vs e) (compE₁ Vs e₁) else (compE₁ Vs e₂)  
| compE1 Vs (while (e) c) = while (compE₁ Vs e) (compE₁ Vs c)  
| compE1 Vs (try e) = throw (compE₁ Vs e)  
| compE1 Vs (try e₁ catch(C V) e₂) = 
  try(compE₁ Vs e₁) catch(C (size Vs)) (compE₁ (Vs@[V]) e₂) 
| compE \_1 Vs [] = []  
| compE \_1 Vs (e # es) = compE1 Vs e # compE₁ Vs es

lemma [simp]: compE₁ Vs es = map (compE₁ Vs) es

primrec fin₁ :: expr ⇒ expr1 where
fin₁(Val v) = Val v  
| fin₁(throw e) = throw(fin₁ e)

lemma comp-final: final e ⇒ compE₁ Vs e = fin₁ e

lemma [simp]:
\[ \forall Vs. \text{max-vars} (compE_1 \ Vs \ e) = \text{max-vars} \ e \]

and \[ \forall Vs. \text{max-vars} (compEs_1 \ Vs \ es) = \text{max-vars} \ es \]

Compiling programs:

**definition** compP \_1 :: J-prog \Rightarrow J_3-prog

where

\[ \text{compP}_1 \equiv \text{compP} (\lambda (\text{pns, body}). \text{compE}_1 (\text{this} \# \text{pns}) \text{ body}) \]

end

### 5.5 Correctness of Stage 1

**theory** Correctness1

imports JIWellForm 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 (\text{new} C) i = True |

unmod (\text{Cast} C e) i = \text{unmod} e i |

unmod (\text{Val} v) i = True |

unmod (\text{e1} <bop> e2) i = (\text{unmod} e_1 i \land \text{unmod} e_2 i) |

unmod (\text{Var} i) j = True |

unmod (\text{i:=e}) j = (\text{i} \neq j \land \text{unmod} e j) |

unmod (\text{e:F}\{D\}) i = \text{unmod} e i |

unmod (\text{e:M}(es)) i = (\text{unmod} e i \land \text{unmods} es i) |

unmod \{j:T; e\} i = \text{unmod} e i |

unmod (\text{e1;e2}) i = (\text{unmod} e_1 i \land \text{unmod} e_2 i) |

unmod (\text{if} \ (\text{e}) e_1 \text{ else } e_2) i = (\text{unmod} e i \land \text{unmod} e_1 i \land \text{unmod} e_2 i) |

unmod (\text{while} \ (\text{e}) \ e) i = (\text{unmod} e i \land \text{unmod} e i) |

unmod (\text{throw} e) i = \text{unmod} e i |

unmod (\text{try} e_1 \text{ catch}(C i) e_2) j = (\text{unmod} e_1 j \land (\text{if} i=j \text{ then False else } \text{unmod} e_2 j)) |

unmods (\text{[]}) i = True |

unmods (\text{e#es}) i = (\text{unmod} e i \land \text{unmods} es i)

**lemma** hidden-unmod: \[ \forall Vs. \text{hidden} Vs i \Rightarrow \text{unmod} (\text{compE}_1 \ Vs \ e) i \text{ and} \]

\[ \forall Vs. \text{hidden} Vs i \Rightarrow \text{unmods} (\text{compEs}_1 \ Vs \ es) i \]

**lemma** eval1-preserves-unmod:

\[ \[ P \vdash_1 (\text{e}, (h, ls)) \Rightarrow (\text{e}', (h', ls')); \text{unmod} e \ i; i < \text{size} \ ls \ \Rightarrow \ls ! i = \ls' ! i \]

and \[ \[ P \vdash_1 (\text{es}, (h, ls)) \Rightarrow (\text{es}', (h', ls')); \text{unmods} es \ i; i < \text{size} \ ls \ \Rightarrow \ls ! i = \ls' ! i \]

**lemma** LAss-lem:

\[ \{ x \in \text{set} \ xs; \text{size} \ xs \leq \text{size} \ ys \} \Rightarrow m_1 \subseteq m_2 (x \mapsto y) \subseteq m_1 (x \mapsto y) \subseteq m_2 (x \mapsto y)s | y = y \} \]

**lemma** Block-lem:

**fixes** l :: \'a \Rightarrow \'b

assumes \[ \emptyset : l \subseteq \emptyset [Vs \mapsto] \]

and \[ l : l' \subseteq l [Vs \mapsto] \]

and
and hidden: \( V \in \text{set} \ Vs \implies \text{l} \ ! \ \text{last-index} \ Vs \ V = \text{l} \ ! \ \text{last-index} \ Vs \ V \)
and size: \( \text{size} \ \text{l} \ V = \text{size} \ \text{l} \ V \)
\( \text{size} \ Vs < \text{size} \ \text{l} \ V \)
shows \( l'((V := l \ V)) \subseteq \text{m} \ \{ \text{eval} \ Vs \ V \} \)

The main theorem:

**theorem assumes** \( \text{wf}: \text{wuf-J-prog} \ P \)
**shows** \( \text{eval}_{1,2} - \text{eval}_{L}: P + \{ e,(h,l) \} \implies (e',(h',l')) \)
\( \implies (\bigwedge V \ \text{Ts} \ U. \ [ e :\text{U} ; \text{size} \ \text{Ts} = \text{size} \ Vs \ ] \)
\( \implies \text{compP} f P, Ts \vdash_{1} \text{compE}_{1} V s e :\text{U} \)
and \( \bigwedge V \ \text{Ts} \ Us. \ [ e :\text{Us} ; \text{size} \ \text{Ts} = \text{size} \ Vs \ ] \)
\( \implies \text{compP} f P, Ts \vdash_{1} \text{compEs}_{1} V s e :\text{Us} \)

and the correct block numbering:

**lemma B**: \( \bigwedge V \ n. \ \text{size} \ Vs = n \implies B (\text{compE}_{1} V s e) \)
and \( B s: \bigwedge V \ n. \ \text{size} \ Vs = n \implies B s (\text{compEs}_{1} V s e) \)

The main complication is preservation of definite assignment \( \mathcal{D} \).

**lemma image-last-index**: \( A \subseteq \text{set}(x \ @ [x]) \implies \text{last-index} (x \ @ [x]) \cdot A = \)
\( (\text{if} \ x \in A \ \text{then} \ \text{insert} \ (\text{size} \ x) \ (\text{last-index} \ x) \cdot (A \setminus \{x\})) \ \text{else} \ \text{last-index} \ x \cdot A) \)

**lemma A-compE1-None[simp]**:
\( \bigwedge V \ A e = \text{None} \implies A (\text{compE}_{1} V s e) = \text{None} \)
and \( \bigwedge V \ A s e = \text{None} \implies As (\text{compEs}_{1} V s e) = \text{None} \)

**lemma A-compE1**:
\( \bigwedge A V s. \ A e = [A] ; f u e \subseteq \text{set} \ Vs \ [ e : \text{set} \ Vs ] \implies A (\text{compE}_{1} V s e) = [\text{last-index} V s \cdot A] \)
and \( \bigwedge A V s. \ [ A \subseteq \text{set} \ Vs ; f u e \subseteq \text{set} \ Vs ] \implies As (\text{compEs}_{1} V s e) = [\text{last-index} V s \cdot A] \)

**lemma D-None[iff]**: \( \mathcal{D} (e,:\text{a exp}) \ \text{None} \ \text{and} \ [\text{iff}]: \mathcal{D} s (e,:\text{a exp list}) \ \text{None} \)

**lemma D-last-index-compE1**:
\( \bigwedge A V s. \ [ A \subseteq \text{set} \ Vs ; f u e \subseteq \text{set} \ Vs ] \implies \)
\( \mathcal{D} e [A] \implies \mathcal{D} (\text{compE}_{1} V s e) [\text{last-index} V s \cdot A] \)
and \( \bigwedge A V s. \ [ A \subseteq \text{set} \ Vs ; f u e \subseteq \text{set} \ Vs ] \implies \)
\( \mathcal{D} s e [A] \implies \mathcal{D} s (\text{compEs}_{1} V s e) [\text{last-index} V s \cdot A] \)

**lemma last-index-image-set**: \( \text{distinct} \ x s \implies \text{last-index} x s \cdot \text{set} \ x s = \{ .. \cdot \text{size} \ x s \} \)

**lemma D-compE1**:
\( \bigwedge \mathcal{D} e [\text{set} \ Vs] ; f u e \subseteq \text{set} \ Vs ; \text{distinct} \ [\text{set} \ Vs] \implies \mathcal{D} (\text{compE}_{1} V s e) [.. \cdot \text{length} \ Vs] \)
lemma D-compE_1':
assumes D e [set(V(#Vs))] and fe e \subseteq set(V(#Vs)) and distinct(V(#Vs))
shows D (compE_1 (V(#Vs) e) [[..length Vs]]

lemma compP_1-pres-wf: wf-J-prog P \implies wf-J_1-prog (compP_1 P)

end

5.6 Compilation Stage 2

theory Compiler2
imports PCompiler J1 ../JVM/JVMExec
begin

primrec compE_2 :: expr_1 \Rightarrow instr list
and compEs_2 :: instr list \Rightarrow instr list where
  compE_2 (new C) = [New C]
  | compE_2 (Cast C e) = compE_2 e @ [Checkcast C]
  | compE_2 (Val v) = [Push v]
  | compE_2 (e | bop | e_2) = compE_2 e_1 @ compE_2 e_2 @
  
  (case bop of Eq \Rightarrow [CmpEq]
    | Add \Rightarrow [IAdd])
  | compE_2 (Var i) = [Load i]
  | compE_2 (i := e) = compE_2 e @ [Store i, Push Unit]
  | compE_2 (e | F\{D\}) = compE_2 e @ [Getfield F D]
  | compE_2 (e_1 | F\{D\} := e_2) =
    compE_2 e_1 @ compE_2 e_2 @ [Putfield F D, Push Unit]
  | compE_2 (i | T; e) = compE_2 e
  | compE_2 (e_1; e_2) = compE_2 e_1 @ [Pop] @ compE_2 e_2
  | compE_2 (if (e) e_1 else e_2) =
    (let end = compE_2 e;
     thn = compE_2 e_1;
     els = compE_2 e_2;
     test = IfFalse (int(size thn + 2));
     thnex = Goto (int(size els + 1))
    in end @ [test] @ thn @ [thnex] @ els)
  | compE_2 (while (e) c) =
    (let end = compE_2 e;
     bdy = compE_2 c;
     test = IfFalse (int(size bdy + 3));
     loop = Goto (size bdy + size end + 2))
    in end @ [test] @ bdy @ [Pop] @ [loop] @ [Push Unit])
  | compE_2 (throw e) = compE_2 e @ [instr.Throw]
  | compE_2 (try e_1 catch (C i) e_2) =
    (let catch = compE_2 e_2
     in compE_2 e_1 @ [Goto (size catch) + 2], Store i] @ catch)
  | compEs_2 [] = []
  | compEs_2 (e(#es) = compE_2 e @ compEs_2 es

Compilation of exception table. Is given start address of code to compute absolute addresses necessary in exception table.
primrec \textit{compxE}_2 :: \textit{expr}_1 \Rightarrow \textit{pc} \Rightarrow \textit{nat} \Rightarrow \textit{ex-table}
\textbf{and} \textit{compxE}_2 :: \textit{expr}_1 \textit{list} \Rightarrow \textit{pc} \Rightarrow \textit{nat} \Rightarrow \textit{ex-table} \textbf{where}

\textit{compxE}_2 (\textit{new C}) \textit{pc} \textit{d} = []
| \textit{compxE}_2 (\textit{Cast C e}) \textit{pc} \textit{d} = \textit{compxE}_2 \textit{pc} \textit{d}
| \textit{compxE}_2 (\textit{Val v}) \textit{pc} \textit{d} = []
| \textit{compxE}_2 (\textit{e1 <lop> e2}) \textit{pc} \textit{d} =
\textit{compxE}_2 \textit{e1} \textit{pc} \textit{d} \@ \textit{compxE}_2 \textit{e2} (\textit{pc} + \textit{size} (\textit{compxE}_2 \textit{e1})) (\textit{d}+1)
| \textit{compxE}_2 (\textit{Var i}) \textit{pc} \textit{d} = []
| \textit{compxE}_2 (\textit{i:=c}) \textit{pc} \textit{d} = \textit{compxE}_2 \textit{e2} \textit{pc} \textit{d}
| \textit{compxE}_2 (\textit{e.F\{D\}}) \textit{pc} \textit{d} = \textit{compxE}_2 \textit{pc} \textit{d}
| \textit{compxE}_2 (\textit{e1.F\{D\}} := \textit{e2}) \textit{pc} \textit{d} =
\textit{compxE}_2 \textit{e1} \textit{pc} \textit{d} \@ \textit{compxE}_2 \textit{e2} (\textit{pc} + \textit{size} (\textit{compxE}_2 \textit{e1})) (\textit{d}+1)
| \textit{compxE}_2 (\textit{e-M\{es\}}) \textit{pc} \textit{d} =
\textit{compxE}_2 \textit{pc} \textit{d} \@ \textit{compxE}_2 \textit{es} (\textit{pc} + \textit{size} (\textit{compxE}_2 \textit{e})) (\textit{d}+1)
| \textit{compxE}_2 (\textit{i:T; e}) \textit{pc} \textit{d} = \textit{compxE}_2 \textit{e} \textit{pc} \textit{d}
| \textit{compxE}_2 (\textit{e1;:e2}) \textit{pc} \textit{d} =
\textit{compxE}_2 \textit{e1} \textit{pc} \textit{d} \@ \textit{compxE}_2 \textit{e2} (+\textit{size} (\textit{compxE}_2 \textit{e1})+1) \textit{d}
| \textit{compxE}_2 (\textit{if} (\textit{e} \textit{e1} \textit{else} \textit{e2}) \textit{pc} \textit{d} =
\textit{let} \textit{pc1} = \textit{pc} + \textit{size} (\textit{compxE}_2 \textit{e})+1;
\textit{pc2} = \textit{pc1} + \textit{size} (\textit{compxE}_2 \textit{e1})+1
\textit{in} \textit{compxE}_2 \textit{e} \textit{pc} \textit{d} \@ \textit{compxE}_2 \textit{e1} \textit{pc1} \textit{d} \@ \textit{compxE}_2 \textit{e2} \textit{pc2} \textit{d}
| \textit{compxE}_2 (\textit{while} (\textit{b} \textit{e}) \textit{pc} \textit{d} =
\textit{compxE}_2 \textit{b} \textit{pc} \textit{d} \@ \textit{compxE}_2 \textit{e} (\textit{pc}+\textit{size} (\textit{compxE}_2 \textit{b})+1) \textit{d}
| \textit{compxE}_2 (\textit{throw} \textit{e}) \textit{pc} \textit{d} = \textit{compxE}_2 \textit{e} \textit{pc} \textit{d}
| \textit{compxE}_2 (\textit{try} \textit{e} \textit{catch} (\textit{C i} \textit{e2}) \textit{pc} \textit{d} =
\textit{let} \textit{pc1} = \textit{pc} + \textit{size} (\textit{compxE}_2 \textit{e1})
\textit{in} \textit{compxE}_2 \textit{e1} \textit{pc} \textit{d} \@ \textit{compxE}_2 \textit{e2} (\textit{pc1}+2) \textit{d} \@ [\textit{pc}, \textit{pc1}, \textit{pc1}+1, \textit{d}]
| \textit{compxE}_2 [] \textit{pc} \textit{d} = []
| \textit{compxE}_2 (\textit{e#es}) \textit{pc} \textit{d} = \textit{compxE}_2 \textit{e} \textit{pc} \textit{d} \@ \textit{compxE}_2 \textit{es} (\textit{pc}+\textit{size} (\textit{compxE}_2 \textit{e})) (\textit{d}+1)

primrec \textit{max-stack} :: \textit{expr}_1 \Rightarrow \textit{nat}
\textbf{and} \textit{max-stacks} :: \textit{expr}_1 \textit{list} \Rightarrow \textit{nat} \textbf{where}

\textit{max-stack} (\textit{new C}) = 1
| \textit{max-stack} (\textit{Cast C e}) = \textit{max-stack} \textit{e}
| \textit{max-stack} (\textit{Val v}) = 1
| \textit{max-stack} (\textit{e1 <lop> e2}) = \textit{max} (\textit{max-stack} \textit{e1}) (\textit{max-stack} \textit{e2}) + 1
| \textit{max-stack} (\textit{Var i}) = 1
| \textit{max-stack} (\textit{i:=c}) = \textit{max-stack} \textit{e}
| \textit{max-stack} (\textit{e.F\{D\}}) = \textit{max-stack} \textit{e}
| \textit{max-stack} (\textit{e1.F\{D\}} := \textit{e2}) = \textit{max} (\textit{max-stack} \textit{e1}) (\textit{max-stack} \textit{e2}) + 1
| \textit{max-stack} (\textit{e-M\{es\}}) = \textit{max} (\textit{max-stack} \textit{e}) (\textit{max-stacks} \textit{es}) + 1
| \textit{max-stack} (\textit{i:T; e}) = \textit{max-stack} \textit{e}
| \textit{max-stack} (\textit{e1;:e2}) = \textit{max} (\textit{max-stack} \textit{e1}) (\textit{max-stack} \textit{e2})
| \textit{max-stack} (\textit{if} (\textit{e} \textit{e1} \textit{else} \textit{e2}) =
\textit{max} (\textit{max-stack} \textit{e}) (\textit{max-stack} \textit{e1}) (\textit{max-stack} \textit{e2})
| \textit{max-stack} (\textit{while} (\textit{e} \textit{c}) = \textit{max} (\textit{max-stack} \textit{e}) (\textit{max-stack} \textit{c})
| \textit{max-stack} (\textit{throw} \textit{e}) = \textit{max-stack} \textit{e}
| \textit{max-stack} (\textit{try} \textit{e} \textit{catch} (\textit{C i} \textit{e2}) = \textit{max} (\textit{max-stack} \textit{e1}) (\textit{max-stack} \textit{e2})

| \textit{max-stacks} [] = 0
| \textit{max-stacks} (\textit{e#es}) = \textit{max} (\textit{max-stack} \textit{e}) (1 + \textit{max-stacks} \textit{es})

lemma \textit{max-stack1} : 1 \leq \textit{max-stack} \textit{e}
definition \textit{compMb}_2 :: \textit{expr} \Rightarrow \textit{jvm-method}
where
\( \textit{compMb}_2 \equiv \lambda \textit{body}. \)
let \( \textit{ins} = \textit{compE}_2 \textit{body} \odot [\text{Return}] \);
\( \textit{xt} = \textit{compxE}_2 \textit{body} 0 0 \)
in \( \text{max-stack} \textit{body}, \text{max-vars} \textit{body}, \textit{ins}, \textit{xt} \)

definition \textit{compP}_2 :: \textit{J}_1\text{-prog} \Rightarrow \textit{jvm-prog}
where
\( \textit{compP}_2 \equiv \textit{compP} \textit{compMb}_2 \)

lemma \textit{compMb}_2 [simp]:
\( \textit{compMb}_2 \ e = (\text{max-stack} \ e, \text{max-vars} \ e, \textit{compE}_2 \ e \odot [\text{Return}], \textit{compxE}_2 \ e 0 0) \)

end

5.7 Correctness of Stage 2

theory \textit{Correctness2}
imports \texttt{\sim\textnormal{/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 \textit{before} :: \textit{jvm-prog} \Rightarrow \textit{cname} \Rightarrow \textit{mname} \Rightarrow \textit{nat} \Rightarrow \textit{instr} \textit{list} \Rightarrow \textit{bool}
(\( (\_,-,-,/-,\_)/\Rightarrow \_ \_ [51,0,0,0,51] 50) \) where
\( P,C,M,pc \Rightarrow is \leftarrow \text{prefixeq} \ (\text{drop} \ pc \ \text{instrs-of} \ P \ C \ M) \)

definition \textit{at} :: \textit{jvm-prog} \Rightarrow \textit{cname} \Rightarrow \textit{mname} \Rightarrow \textit{nat} \Rightarrow \textit{instr} \Rightarrow \textit{bool}
(\( (\_,-,-,/-,\_)/\Rightarrow \_ \_ [51,0,0,0,51] 50) \) where
\( P,C,M,pc \Rightarrow i \leftarrow (\exists \ is. \ \text{drop} \ pc \ (\text{instrs-of} \ P \ C \ M) = i \# is) \)

lemma [simp]: \( P,C,M,pc \Rightarrow [] \)

lemma [simp]: \( P,C,M,pc \Rightarrow (i \# is) = (P,C,M,pc \Rightarrow i \land P,C,M,pc + 1 \Rightarrow is) \)

lemma [simp]: \( P,C,M,pc \Rightarrow (is_1 @ is_2) = (P,C,M,pc \Rightarrow is_1 \land P,C,M,pc + \text{size} \ is_1 \Rightarrow is_2) \)

lemma [simp]: \( P,C,M,pc \Rightarrow i \Rightarrow \text{instrs-of} \ P \ C \ M \ ! \ pc = i \)

lemma \textit{beforeM}:
\( P \vdash C \text{ sees } M; Ts \Rightarrow T = \text{body in } D \Rightarrow \)
\( \textit{compP}_2 P,D,M,0 \Rightarrow \textit{compE}_2 \textit{body} \odot [\text{Return}] \)

This lemma executes a single instruction by rewriting:

lemma [simp]:
\( P,C,M,pc \Rightarrow \textit{instr} \Rightarrow \)
\( (P \vdash (\text{None},h,(\text{vs,ls,C,M,pc}) \# \text{frs}) \text{-jvm} \Rightarrow \sigma') = \)
\( ((\text{None},h,(\text{vs,ls,C,M,pc}) \# \text{frs}) = \sigma' \lor \)

This

This

This

This

This

This

This

This

This

This

This

This

This
(∃σ. exec(P,(None, h, (vs,ls,C,M,pc) # frs)) = Some σ ∧ P ⊢ σ –jvm→ σ′)

5.7.2 Exception tables

definition pcs :: ex-table ⇒ nat set

where
pcs xt = \((f,t,C,h,d) \in set xt. \{f ..< t\}\)

lemma pcs-subset:
shows ∀pc d. pcs(compxE2 e pc d) ⊆ \{pc..<pc+size(compE2 e)\}
and ∀pc d. pcs(compxE2 es pc d) ⊆ \{pc..<pc+size(compEs2 es)\}

lemma [simp]: pcs [] = {}

lemma [simp]: pcs (x#xt) = \{fst x ..< fst(snd x)\} ∪ pcs xt

lemma [simp]: pcs(xt1 @ xt2) = pcs xt1 ∪ pcs xt2

lemma [simp]: pc < pc0 ∨ pc0+size(compE2 e) ≤ pc ⇒ pc /∈ pcs(compE2 e pc0 d)

lemma [simp]: pc < pc0 ∨ pc0+size(compEs2 es) ≤ pc ⇒ pc /∈ pcs(compEs2 es pc0 d)

lemma [simp]: pc1 + size(compE2 e1) ≤ pc2 ⇒ pcs(compE2 e1 pc1 d1) ∩ pcs(compE2 e2 pc2 d2) = {}

lemma [simp]: pc1 + size(compE2 e) ≤ pc2 ⇒ pcs(compE2 e pc1 d1) ∩ pcs(compEs2 es pc2 d2) = {}

lemma [simp]:

pc /∈ pcs xt0 ⇒ match-ex-table P C pc (xt0 @ xt1) = match-ex-table P C pc xt1

lemma [simp]: \[ x ∈ set xt; pc /∈ pcs xt \] ⇒ ¬ match-ex-entry P D pc x

lemma [simp]:

assumes xe: xe ∈ set(compxE2 e pc d) and outside: pc′ < pc ∨ pc+size(compE2 e) ≤ pc′
shows ¬ match-ex-entry P C pc′ xe

lemma [simp]:

assumes xe: xe ∈ set(compEs2 es pc d) and outside: pc′ < pc ∨ pc+size(compEs2 es) ≤ pc′
shows ¬ match-ex-entry P C pc′ xe

lemma match-ex-table-app[simp]:
\forall zte ∈ set xt1, ¬ match-ex-entry P D pc zte ⇒
match-ex-table P D pc (xt1 @ xt) = match-ex-table P D pc xt

lemma [simp]:
\forall x ∈ set xtab, ¬ match-ex-entry P C pc x ⇒
match-ex-table P C pc xtab = None

lemma match-ex-entry:
match-ex-entry P C pc (start, end, catch-type, handler) =
(start ≤ pc ∧ pc < end ∧ P ⊢ C ≈* catch-type)

definition caught :: jvm-prog ⇒ pc ⇒ heap ⇒ addr ⇒ ex-table ⇒ bool where
matches-ex-entry \( P \) (cname-of \( h \ a \) pc entry)

**definition** beforex :: jvm-prog \( \Rightarrow \) cname \( \Rightarrow \) mname \( \Rightarrow \) ex-table \( \Rightarrow \) nat set \( \Rightarrow \) nat \( \Rightarrow \) bool

\( ((2,\sim,\sim \triangleright / - / \sim / - \ sim) [51,0,0,0,51] 50) \) where

\( P,C,M \triangleright xt / I,d \iff (\exists xt_0 \ xt_1. \ ex-table-of \ P \ M = xt_0 \ @ \ xt \ @ \ xt_1 \land pcs xt_0 \cap I = \{\} \land pcs xt \subseteq I \land \forall pc \in I. \forall C \ pc \ d', \ ex-table-of \ P \ pc \ xt_1 = [(pc',d')] \longrightarrow d' \leq d) \)

**definition** dummyx :: jvm-prog \( \Rightarrow \) cname \( \Rightarrow \) mname \( \Rightarrow \) ex-table \( \Rightarrow \) nat set \( \Rightarrow \) nat \( \Rightarrow \) bool

\( ((2,\sim,\sim \triangleright / - / \sim / - \ sim) [51,0,0,0,51] 50) \) where

\( P,C,M \triangleright xt/1,d \iff P,C,M \triangleright xt/1,d \)

**lemma** beforexD1: \( P,C,M \triangleright xt / I,d \implies pcs xt \subseteq I \)

**lemma** beforex-mono: \( [P,C,M \triangleright xt/I,d'; d' \leq d] \implies P,C,M \triangleright xt/I,d \)

**lemma** [simpl]: \( P,C,M \triangleright xt/I,d \implies P,C,M \triangleright xt/I,Suc \ d \)

**lemma** beforex-append[simpl]:

\( pcs xt_1 \cap pcs xt_2 = \{\} \implies P,C,M \triangleright xt_1 \cap xt_2 / I,d = (P,C,M \triangleright xt_1/I \neg pcs xt_2,d \land P,C,M \triangleright xt_2/I \neg pcs xt_1,d \land P,C,M \triangleright xt_1 \cap xt_2 / I,d) \)

**lemma** beforex-appendD1:

\( [P,C,M \triangleright xt_1 \cap xt_2 @ [(f,t,D,h,d)] / I,d; pcs xt_1 \subseteq J; J \subseteq I; J \cap pcs xt_2 = \{\}] \implies P,C,M \triangleright xt_1 / J,d \)

**lemma** beforex-appendD2:

\( [P,C,M \triangleright xt_1 \cap xt_2 @ [(f,t,D,h,d)] / I,d; pcs xt_2 \subseteq J; J \subseteq I; J \cap pcs xt_1 = \{\}] \implies P,C,M \triangleright xt_2 / J,d \)

**lemma** beforeM:

\( P \vdash C \ sees \ M : Ts \rightarrow T = \ body \ in \ D \implies \ \compP_2 \ P,D,M \vdash \ \comp\ E_2 \ body \ 0 \ 0 / \{..<\size(\comp\ E_2 \ body)\},0 \)

**lemma** match-ex-table-SomeD2:

\( [\ \match-ex-table \ P \ D \ pc \ (ex-table-of \ P \ C \ M) = [(pc',d')] ; P,C,M \triangleright xt/I,d; \forall x \in set \ xt. \neg matches-ex-entry \ P \ D \ pc \ x; pc \in I \] \implies d' \leq d \)

**lemma** match-ex-table-SomeD1:

\( [\ \match-ex-table \ P \ D \ pc \ (ex-table-of \ P \ C \ M) = [(pc',d')] ; P,C,M \triangleright xt/I,d; pc \in I; pc \notin pcs xt \] \implies d' \leq d \)

### 5.7.3 The correctness proof

**definition**

\( handle :: jvm-prog \Rightarrow cname \Rightarrow mname \Rightarrow addr \Rightarrow heap \Rightarrow val \ list \Rightarrow val \ list \Rightarrow nat \Rightarrow frame \ list \Rightarrow jvm-state \)
handle \( P \ C \ M \ a \ h \ vs \ ls \ pc \ frs = \text{find-handler} \ P \ a \ h \ ((vs,ls,C,M,pc) \neq frs) \)

**lemma** handle-Cons:

\[
\begin{align*}
&P,C,M \triangleright xt/I,d; \ d \leq \text{size} \ vs; \ pc \in I; \\
&\quad \forall x \in \text{set} \ xt. \ \neg \text{matches-ex-entry} \ P \ (\text{name-of} \ h \ xa) \ pc \ x \end{align*}
\]

handle \( P \ C \ M \ xa \ h \ (v \neq vs) \ ls \ pc \ frs = \text{handle} \ P \ C \ M \ xa \ h \ vs \ ls \ pc \ frs
\]

**lemma** handle-append:

\[
\begin{align*}
&P,C,M \triangleright xt/I,d; \ d \leq \text{size} \ vs; \ pc \in I; \ pc \notin \text{pcs} \ xt \] \\
&\quad \Rightarrow \hspace{1em} \text{handle} \ P \ C \ M \ xa \ h \ (vs \ @ vs) \ ls \ pc \ frs = \text{handle} \ P \ C \ M \ xa \ h \ vs \ ls \ pc \ frs
\]

**lemma** aux-isin[simp]: \([ \ B \subseteq A; \ a \in B \] \implies a \in A\)

**lemma** fixes \( P_1 \) defines [simp]: \( P \equiv \text{compP}_2 \ P_1 \)

**shows** Jcc:

\[
P_1 \vdash_1 (e,(h_0,ls_0)) \Rightarrow (\langle f,(h_1,ls_1) \rangle
\]

\(\langle \forall \ C \ M \ pc \ v \ xa \ vs \ frs \ I . \\
\quad \exists e \ pc \ h_1 \ xa \ (\text{compE}_2 \ e \ pc \ (\text{size} \ vs)) \wedge \\
\quad P \vdash (\text{None},h_0,(vs,ls_0,C,M,pc)\#frs) \rightarrow_{jvm} \hspace{1em} \text{handle} \ P \ C \ M \ xa \ h_1 \ vs \ ls_1 \ pc \ frs))))
\]

**and** \( P_1 \vdash_1 (es,(h_0,ls_0)) \Rightarrow (\langle fs,(h_1,ls_1) \rangle
\]

\(\langle \forall \ C \ M \ pc \ v \ xa \ es' \ vs \ frs \ I . \\
\quad \exists es \ PC \ M \ pc \ es' \ PC \ M \ pc \ (\text{size} \ vs) \ I . \hspace{1em} \text{size} \ vs; \\
\quad (fs = \text{map} \ Val \ ws \rightarrow \\
\quad P \vdash (\text{None},h_0,(vs,ls_0,C,M,pc)\#frs) \rightarrow_{jvm} \hspace{1em} \text{handle} \ P \ C \ M \ xa \ h_1 \ vs \ ls_1 \ pc \ frs))))
\]

**lemma** atLeast0AtMost[simp]: \( \{0::\text{nat}..n\} = \{..n\} \)

by auto

**lemma** atLeast0LessThan[simp]: \( \{0::\text{nat}..<n\} = \{..<n\} \)

by auto

**fun** exception :: 'a \ exp \ ⇒ \ addr \ option \ where

\[
\begin{align*}
&\text{exception} \ (\text{Throw} \ a) = \text{Some} \ a \\
&\text{exception} \ e = \text{None}
\end{align*}
\]

**lemma** comp2_correct:

**assumes** method: \( P_1 \vdash C \ sees \ M; Ts \rightarrow T = \text{body} \ in \ C \)

**and** eval: \( P_1 \vdash_1 \langle \text{body},(h,ls) \rangle \Rightarrow \langle e',(h',ls') \rangle \)
shows \( \text{compP}_2 \ P_1 \vdash (\text{None}, h, [(\text{ls}, C, M, 0)])) \rightarrow \text{jvm} \rightarrow (\text{exception } e', h', []) \)

end

### 5.8 Combining Stages 1 and 2

theory Compiler
imports Correctness1 Correctness2
begin

definition \( J2JVM : \ J\text{-prog} \Rightarrow \ jvm\text{-prog} \)
where
\( J2JVM \equiv \text{compP}_2 \circ \text{compP}_1 \)

theorem \( \text{comp-correct} \):
assumes \( \text{wwf-J-prog } P \)
and \( \text{method: } P \vdash C \) sees \( M \):
\( \text{Ts} \rightarrow T = (pns, \text{body}) \in C \)
and \( \text{eval: } P \vdash \langle \text{body}, h, [\text{this} \# pns [\mapsto \rightarrow \text{vs}]] \rangle \Rightarrow \langle e', h', [] \rangle \)
and \( \text{sizes: } \text{size } \text{vs} = \text{size } \text{pns} + 1 \quad \text{size } \text{rest} = \text{max-vars} \text{ body} \)
shows \( J2JVM \ P \vdash (\text{None}, h, [(\text{vs} @ \text{rest}, C, M, 0)]) - \text{jvm} \rightarrow (\text{exception } e', h', []) \)

end

### 5.9 Preservation of Well-Typedness

theory TypeComp
imports Compiler ../ BV / BVSpec
begin

locale TC0 =
  fixes \( P : J_1\text{-prog} \) and \( \text{mxl :: nat} \)
begin

definition \( \text{ty } E \ e = (\text{THE } T. \ P, E \vdash_1 e :: T) \)

definition \( \text{ty}_l \ E \ A' = \text{map} \ (\lambda i. \text{if } i \in A' \land i < \text{size } E \text{ then } \text{OK}(E!i) \text{ else } \text{Err}) \ [0..<\text{mxl}] \)

definition \( \text{ty}_l' \ ST \ E \ A = (\text{case } A \text{ of None } \Rightarrow \text{None } | \ [A'] \Rightarrow \text{Some}(ST, \text{ty}_l \ E \ A')) \)

definition \( \text{after } E \ A \ ST \ e = \text{ty}_l' \ (\text{ty } E \ e \# ST) \ E \ (A \uplus A \ e) \)

end

lemma (in TC0) \( \text{ty-def2 } [\text{simp}]: P, E \vdash_1 e :: T \implies \text{ty } E \ e = T \)
lemma (in TC0) \( [\text{simp}]: \text{ty}_l' \ ST \ E \ \text{None} = \text{None} \)
lemma (in TC0) \( \text{ty}_l\text{-app-diff}[\text{simp}]: \)
\( \text{ty}_l (E@[T]) (A - \{\text{size } E\}) = \text{ty}_l \ E \ A \)

lemma (in TC0) \( \text{ty}_l\text{-app-diff}[\text{simp}]: \)
\( \text{ty}_l' \ ST \ (E \otimes [T]) (A \odot \text{size } E) = \text{ty}_l' \ ST \ E \ A \)

lemma (in TC0) \( \text{ty}_l\text{-antimono}: \)
\( A \subseteq A' \implies P \vdash \text{ty}_l \ E \ A' [\leq] \text{ty}_l \ E \ A \)
lemma (in TC0) tyL'-anti mono:
A ⊆ A' ⇒ P ⊩ tyL' ST E [A'] ⊆' tyL' ST E [A]

lemma (in TC0) tyL-env-anti mono:
P ⊩ tyL (E@[T]) A [≤τ] tyL E A

lemma (in TC0) tyL'-env-anti mono:
P ⊩ tyL' ST (E@[T]) A ≤' tyL' ST E A

lemma (in TC0) tyL'-incr:
P ⊩ tyL' ST (E@[T]) [insert (size E) A] ≤' tyL' ST E A

lemma (in TC0) tyL-incr:
P ⊩ tyL (E@[T]) (insert (size E) A) [≤τ] tyL E A

lemma (in TC0) tyL-in-types:
set E ⊆ types P ⇒ tyL E A ∈ list mzl (err (types P))
locale TC1 = TC0
begin

primrec compT :: ty list ⇒ nat hyperset ⇒ ty list ⇒ expr1 ⇒ tyL' list and
compTs :: ty list ⇒ nat hyperset ⇒ ty list ⇒ expr list ⇒ tyL' 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 (e1 <bop> e2) =
      (let ST1 = ty E e1 #ST; A1 = A ∪ A e1 in
       compT E A ST e1 @ [after E A ST e1] @
       compT E A1 ST1 e2 @ [after E A1 ST1 e2])
  | compT E A ST (Var i) = []
  | compT E A ST (i := e) = compT E A ST e @
      [after E A ST e, tyL' ST E (A ∪ A e ∪ [[i]])]
  | compT E A ST (e-F{D}) =
      compT E A ST e @ [after E A ST e]
  | compT E A ST (e1,F{D}) := e2 =
      (let ST1 = ty E e1 #ST; A1 = A ∪ A e1; A2 = A1 ∪ A e2 in
       compT E A ST e1 @ [after E A ST e1] @
       compT E A1 ST1 e2 @ [after E A1 ST1 e2] @
       [tyL' ST E A2])
  | compT E A ST {i;T; e} = compT (E@[T]) (A◦i) ST e
  | compT E A ST (e1;;e2) =
      (let A1 = A ∪ A e1 in
       compT E A ST e1 @ [after E A ST e1, tyL' ST E A1] @
       compT E A1 ST e2)
  | compT E A ST (if (e) e1 else e2) =
      (let A0 = A ∪ A e; τ = tyL' ST E A0 in
       compT E A ST e @ [after E A ST e, τ] @
       compT E A0 ST e1 @ [after E A0 ST e1, τ] @
       compT E A0 ST e2)
  | compT E A ST (while (e) c) =
      (let A0 = A ∪ A e; A1 = A0 ∪ A c; τ = tyL' ST E A0 in
       compT E A ST e @ [after E A ST e, τ] @
       compT E A0 ST c @ [after E A0 ST c, tyL' ST E A1, tyL' ST E A0])
| \[ \text{compT} \ E \ A \ \text{ST} \ (\text{throw} \ e) = \text{compT} \ E \ A \ \text{ST} \ e @ \text{[after} \ E \ A \ \text{ST} \ e] \]  
| \[ \text{compT} \ E \ A \ \text{ST} \ (\cdot-M(\text{es})) = \text{compT} \ E \ A \ \text{ST} \ e @ \text{[after} \ E \ A \ \text{ST} \ e] @ \]  
| \[ \text{compTs} \ E \ (A \sqcup A \ e) (ty \ E \ e \ # \ ST) \ es \]  
| \[ \text{compT} \ E \ A \ \text{ST} \ (\text{try} \ e_1 \ \text{catch}(C \ i) \ e_2) = \text{compT} \ E \ A \ \text{ST} \ e_1 \ @ \text{[after} \ E \ A \ \text{ST} \ e_1] @ \]  
| \[ \left\{ ty_i, ty_i', \text{Class} \ C \ # \ ST \right\} E, \ A, ty_i', \text{ST} (E \ # \text{Class} \ C) (A \sqcup [\{i\}]) \ @ \]  
| \[ \text{compT} \ E \ A \ \text{ST} \ (E \ # \text{Class} \ C) (A \sqcup [\{i\}]) \ \text{ST} \ e_2 \]  
| \[ \text{compTs} \ E \ A \ \text{ST} \ [] = [] \]  
| \[ \text{compTs} \ E \ A \ \text{ST} \ (e \ # \ es) = \text{compT} \ E \ A \ \text{ST} \ e @ \text{[after} \ E \ A \ \text{ST} \ e] @ \]  
| \[ \text{compTs} \ E \ (A \sqcup (A \ e)) (ty \ E \ e \ # \ ST) \ es \]  

**definition** \(compT_a:: ty \ list \Rightarrow \text{nat} \ \text{hyperset} \Rightarrow ty \ list \Rightarrow \text{expr}_1 \Rightarrow ty_i' \ list\) \(where\)

\(compT_a \ E \ A \ \text{ST} \ e = \text{compT} \ E \ A \ \text{ST} \ e @ \text{[after} \ E \ A \ \text{ST} \ e]\)

**end**

**lemma** \(compE2\)-not-nil[simp]: \(compE_2 \ e \neq []\)

**lemma (TC1)** \(compT\)-sizes[simp]:

\(\exists E \ A \ \text{ST}. \ \text{size} (\text{compT} \ E \ A \ \text{ST} \ e) = \text{size}(\text{compE}_2 \ e) - 1\)

and \(\exists E \ A \ \text{ST}. \ \text{size}(\text{compTs} \ E \ A \ \text{ST} \ es) = \text{size}(\text{compE}_2 \ es)\)

**lemma (TC1) [simp]:** \(\exists ST \ E. \ \left[ \tau \right] \notin \text{set}(\text{compT} \ E \ \text{None} \ ST \ e)\)

and \(\exists ST \ E. \ \left[ \tau \right] \notin \text{set}(\text{compTs} \ E \ \text{None} \ ST \ es)\)

**lemma (TC0)** \(\text{pair-eq-ty}_i\)-conv:

\([\text{[}ST, LT\text{]}\] \(= ty_i, ST_0 \ E \ A\) =

\((\text{case} \ A \ \text{of} \ \text{None} \Rightarrow \text{False} \mid \text{Some} \ A \Rightarrow (ST = ST_0 \land LT = ty_i \ E \ A))\)

**lemma (TC0)** \(\text{pair-conv-ty}_i:\)

\([\text{[}ST, ty_i \ E \ A\text{]}\] \(= ty_i', ST E \ [A]\)

**lemma (TC1)** \(\text{compT-LT-prefix}:\)

\(\exists E \ A \ ST_0. \ \left[ \text{[}ST, LT\text{]}\right] \in \text{set}(\text{compT} \ E \ A \ ST_0 \ es) \ ; \ B \ e \ (\text{size} \ E) \]

\(\Rightarrow P + \left[ \text{[}ST, LT\text{]}\right] \leq' ty_i' ST \ E \ A\)

and

\(\exists E \ A \ ST_0. \ \left[ \text{[}ST, LT\text{]}\right] \in \text{set}(\text{compTs} \ E \ A \ ST_0 \ es) \ ; \ B \ es \ (\text{size} \ E) \]

\(\Rightarrow P + \left[ \text{[}ST, LT\text{]}\right] \leq' ty_i' ST \ E \ A\)

**lemma [iff]:** \(\text{OK} \ \text{None} \notin \text{states} \ P \ \text{mxs} \ \text{mzl}\)

**lemma (TC0)** \(\text{after-in-states}:\)

\([\ \text{wf-prog} \ P; \ P.E \vdash_1 e :: T; \ \text{set} \ E \subseteq \text{types} \ P; \ \text{set} \ ST \subseteq \text{types} \ P; \]

\(\text{size} \ ST + \text{max-stack} \ e \leq \text{mxs} \]

\(\Rightarrow \text{OK} \ \text{(after} \ E \ A \ \text{ST} \ e) \ \text{in} \ \text{states} \ P \ \text{mxs} \ \text{mzl}\)

**lemma (TC0)** \(\text{OK-ty}_i\)-in-states[simp]:

\([\ \text{set} \ E \subseteq \text{types} \ P; \ \text{set} \ ST \subseteq \text{types} \ P; \ \text{size} \ ST \leq \text{mxs} \]

\(\Rightarrow \text{OK} \ (ty_i' ST \ E \ A) \ \text{in} \ \text{states} \ P \ \text{mxs} \ \text{mzl}\)

**lemma is-class-type-aux: is-class \ P \ C \Rightarrow \text{is-type} \ (\text{Class} \ C)\)

**theorem (TC1)** \(\text{compT-states}:\)

**assumes** \(\text{wf-prog} \ P\)

**shows** \(\exists T \ A \ ST.\)

\([P, E \vdash_1 e :: T; \ \text{set} \ E \subseteq \text{types} \ P; \ \text{set} \ ST \subseteq \text{types} \ P; \]

\(\text{size} \ ST + \text{max-stack} \ e \leq \text{mxs} \]

\(\Rightarrow \text{OK} \ (ty_i' ST \ E \ A) \ \text{in} \ \text{states} \ P \ \text{mxs} \ \text{mzl})\)
size $ST + \text{max-stack } e \leq \text{mxs}; \text{size } E + \text{max-vars } e \leq \text{mxl}$

$\implies$ $\text{OK ' set(\text{compT E A ST e}) \subseteq \text{states P mxs mxl}}$

and $\forall E T s A ST$. 

$\big[ P, E \vdash \text{es} [:] T s; \text{set E} \subseteq \text{types P}; \text{set ST} \subseteq \text{types P};$

$\text{size } ST + \text{max-stacks es} \leq \text{mxs}; \text{size } E + \text{max-vars es} \leq \text{mxl} \big]$

$\implies$ $\text{OK ' set(\text{compTs E A ST es}) \subseteq \text{states P mxs mxl}}$

**Definition** $\text{shift :: nat } \Rightarrow \text{ex-table } \Rightarrow \text{ex-table}$

where

$\text{shift n xt } \equiv \text{map } (\lambda (\text{from, to, C, handler, depth}). (\text{from+n, to+n, C, handler+n, depth})) \text{ xt}$

**Lemma** \[\text{simp}]: $\text{shift 0 xt } = \text{xt}$

**Lemma** \[\text{simp}]: $\text{shift n [] } = []$

**Lemma** \[\text{simp}]: $\text{shift n (xt_1 @ xt_2) } = \text{shift n xt_1 @ shift n xt_2}$

**Lemma** \[\text{simp}]: $\text{shift m (shift n xt) } = \text{shift (m+n) xt}$

**Lemma** \[\text{simp}]: $\text{pcs (shift n xt) } = \{\text{pc+n|pc, pc } \in \text{ pcs xt}\}$

**Lemma** $\text{shift-compxE_2}$:

shows $\forall pc \text{ pc' d}. \text{shift pc (compxE_2 e pc' d) } = \text{compxE_2 e (pc' + pc) d}$

and $\forall pc \text{ pc' d}. \text{shift pc (compxEs_2 es pc' d) } = \text{compxEs_2 es (pc' + pc) d}$

**Lemma** $\text{compxE_2-size-convs}[\text{simp}]:$

shows $n \neq 0 \implies \text{compxE_2 e n d } = \text{shift n (compxE_2 e 0 d)}$

and $n \neq 0 \implies \text{compxEs_2 es n d } = \text{shift n (compxEs_2 es 0 d)}$

**Locale** $TC2 = TC1 +$

**Fixes** $T_r : \text{ty and mxs :: pc}$

begin

**Definition**

$\text{wt-instrs :: instr list } \Rightarrow \text{ex-table } \Rightarrow \text{ty_i' list } \Rightarrow \text{bool}$

$\big((\vdash \cdot \vdash /[:]/ \vdash) [0,0,51] 50\big)$

where

$\vdash is, xt [:] \tau s \leftrightarrow \text{size is } < \text{size } \tau s \land \text{pcs xt } \subseteq \{0..<\text{size is}\} \land$

$(\forall pc < \text{size is}. P, T_r, \text{mxs, size } \tau s, xt \vdash is|pc, pc :: \tau s)$

end

**Notation** $TC2.\text{wt-instrs } ((\vdash \cdot \vdash /[:]/ \vdash) [50,50,50,50,50,50,51] 50)$

**Lemma** \[\text{in TC2}][\text{simp}]: $\tau s \neq [] \implies \vdash [],[] [:] \tau s$

**Lemma** \[\text{simp}]: $\text{eff } i P \text{ pc et None } = []$

**Lemma** $\text{wt-instr-appR}$:

$[ P, T, m, mpe, xt \vdash is|pc, pc :: \tau s;$

$pc < \text{size is}; \text{size is } < \text{size } \tau s; mpe \leq \text{size } \tau s; mpe \leq mpe']$

$\implies P, T, m, mpe', xt \vdash is|pc, pc :: \tau s@\tau s'$

**Lemma** $\text{relevant-entries-shift}[\text{simp}]:$

$\text{relevant-entries P i (pc+n) (shift n xt) } = \text{shift n (relevant-entries P i pc xt)}$

**Lemma** \[\text{simp}]: $\text{xcpt-eff i P (pc+n) } \tau (\text{shift n xt}) =$

$\text{map } (\lambda (pc, \tau). (pc + n, \tau)) (\text{xcpt-eff i P pc } \tau xt)$

**Lemma** \[\text{simp}]:
app_i (i, P, pc, m, T, \tau) \implies 
eff \in P (pc+n) (shift \ n \ xt) (Some \ \tau) = 
map (\lambda (pc, \tau). (pc+n, \tau)) (\eff \in P \ pc \ xt \ (Some \ \tau))

lemma [simp]:
xcept-app \in P (pc+n) mxs (shift \ n \ xt) \tau = xcpt-app \in P \ pc \ mxs \ xt \ \tau

lemma wt-instr-appL:
\[ [P, T, m, mpc, xt] \vdash i, pc :: \tau s; pc < size \tau s; mpc \leq size \tau s \] 
\implies 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 \] 
\implies 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' \] 
\implies P, T, m, mpc, [ ] \vdash i, pc :: \tau s'@\tau s

lemma xcpt-app-pcs:
\[ pc \notin pcs \ xt \implies \text{xcpt-app} \in P \ pc \ mxs \ xt \ \tau \]

lemma xcpt-eff-pcs:
\[ pc \notin pcs \ xt \implies \text{xcpt-eff} \in P \ pc \ \tau \ xt = [] \]

lemma pcs-shift:
\[ pc < n \implies pc \notin pcs \ (shift \ n \ xt) \]

lemma wt-instr-appRx:
\[ [P, T, m, mpc, xt] \vdash i, pc :: \tau s; pc < size \ is; size \ is < size \tau s; mpc \leq size \tau s \] 
\implies P, T, m, mpc, xt @ shift (size \ is) xt' \vdash is!pc, pc :: \tau s

lemma wt-instr-appLx:
\[ [P, T, m, mpc, xt] \vdash i, pc :: \tau s; pc \notin pcs \ xt' \] 
\implies P, T, m, mpc, xt'@xt \vdash i, pc :: \tau s

lemma (in TC2) wt-instrs-extR:
\[ \vdash is, xt :: [:] \ \tau s \implies \vdash is, xt :: [:] \ \tau s @ \tau s' \]

lemma (in TC2) wt-instrs-extL:
\[ \vdash is_1, xt_1 :: [:] \ \tau s_1 @ \tau s_2; \vdash is_2, xt_2 :: [:] \ \tau s_2; size \ \tau s_1 = size \ is_1 \] 
\implies \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:
\[ \vdash is_1, xt_1 :: [:] \ \tau s_2; \vdash is_1, xt_1 :: [:] \ \tau s_1 @ \tau s_2; size \ \tau s_1 = size \ is_1 \] 
\implies \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]:
\[ \vdash is_1, xt_1 :: [:] \ \tau s_1 @ \tau s_2; \vdash is_2, xt_2 :: [:] \ \tau s_2; \] 
size \ \tau s_1 = size \ is_1; prefixeq \ \tau s_2 \ \tau s_2 \] 
\implies \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 $is_1 : \vdash is_1.xt_1 :: \tau s_1 \oplus [\tau]$
assumes $is_2 : \vdash is_2.xt_2 :: \tau \# \tau s_2$
assumes $s : \text{size } \tau s_1 = \text{size } is_1$
shows $\vdash is_1 @ is_2, \ xt_1 @ \text{shift (size } is_1 \) \ xt_2 :: \tau s_1 @ \tau \# \tau s_2$
corollary (in $TC2$) $\text{wt-instrs-app-last}[\text{trans}]$:
$[\vdash is_2.xt_2 :: \tau \# \tau s_2; \vdash is_1.xt_1 :: \tau s_1; \text{last } \tau s_1 = \tau; \text{size } \tau s_1 = \text{size } is_1 + I ]$
$\implies \vdash is_1 @ is_2, \ xt_1 @ \text{shift (size } is_1 \) \ xt_2 :: \tau s_1 @ \tau s_2$
corollary (in $TC2$) $\text{wt-instrs-append-last}[\text{trans}]$:
$[\vdash is.xt :: \tau s; P, T_r, mxs, mpc[] \vdash i, pc :: \tau s; \text{pc = size } is; \text{mpc = size } \tau s; \text{size } is + I < \text{size } \tau s ]$
$\implies \vdash is@[i].xt :: \tau s$
corollary (in $TC2$) $\text{wt-instrs-app2}$:
$[\vdash is_2.xt_2 :: \tau' \# \tau s_2; \vdash is_1.xt_1 :: \tau \# \tau s_1 \oplus [\tau]; \ xt' = xt_1 @ \text{shift (size } is_1 \) \ xt_2; \text{size } \tau s_1 + I = \text{size } is_1 ]$
$\implies \vdash is_1 @ is_2, xt' :: \tau' \# \tau s_1 @ \tau' \# \tau s_2$
corollary (in $TC2$) $\text{wt-instrs-app2-simp}[\text{trans}, \text{simp}]$:
$[\vdash is_2.xt_2 :: \tau' \# \tau s_2; \vdash is_1.xt_1 :: \tau \# \tau s_1 \oplus [\tau]; \text{size } \tau s_1 + I = \text{size } is_1 ]$
$\implies \vdash is_1 @ is_2, \ xt_1 @ \text{shift (size } is_1 \) \ xt_2 :: \tau' \# \tau s_1 @ \tau' \# \tau s_2$
corollary (in $TC2$) $\text{wt-instrs-Cons}[\text{simp}]$:
$[\vdash \tau s \neq []; \vdash [i, [] :: [\tau, \tau]; \vdash is.xt :: \tau \# \tau s ]$
$\implies \vdash i \# is, \text{shift } 1 \ xt :: \tau \# \tau' \# \tau s$
theory $\text{Jinja}$

imports
$J/\text{TypeSafe}$
$J/\text{Annotate}$

$J/\text{execute-WellType}$
$J/\text{execute-Bigstep}$
$JVM/JVMDefensive$
$JVM/JVMListExample$
$BV/\text{BVExec}$
$BV/LBVJVM$
$BV/\text{BVNoTypeErr}$
$BV/\text{BVExample}$
$\text{Compiler/TypeComp}$

begin

end
Bibliography
