Java Source and Bytecode Formalizations in Isabelle: μJava

Gerwin Klein Tobias Nipkow David von Oheimb Cornelia Pusch
Martin Strecker

October 24, 2026

Contents

1 Preface **5**
- 1.1 Introduction . . . 5
- 1.2 Theory Dependencies . . . 7

2 Java Source Language **9**
- 2.1 Some Auxiliary Definitions . . . 9
- 2.2 Java types . . . 10
- 2.3 Class Declarations and Programs . . . 13
- 2.4 Relations between Java Types . . . 14
- 2.5 Java Values . . . 17
- 2.6 Program State . . . 18
- 2.7 Expressions and Statements . . . 20
- 2.8 System Classes . . . 21
- 2.9 Well-formedness of Java programs . . . 21
- 2.10 Well-typedness Constraints . . . 27
- 2.11 Operational Evaluation (big step) Semantics . . . 31
- 2.12 Conformity Relations for Type Soundness Proof . . . 35
- 2.13 Type Safety Proof . . . 39
- 2.14 Example MicroJava Program . . . 41
- 2.15 Example for generating executable code from Java semantics . . . 47

3 Java Virtual Machine **51**
- 3.1 State of the JVM . . . 51
- 3.2 Instructions of the JVM . . . 52
- 3.3 JVM Instruction Semantics . . . 52
- 3.4 Exception handling in the JVM . . . 55
- 3.5 Program Execution in the JVM . . . 55
- 3.6 Example for generating executable code from JVM semantics . . . 56
- 3.7 A Defensive JVM . . . 59

4 Bytecode Verifier **63**
- 4.1 Semilattices . . . 63
- 4.2 The Error Type . . . 67
- 4.3 Fixed Length Lists . . . 71
- 4.4 Typing and Dataflow Analysis Framework . . . 77
- 4.5 Products as Semilattices . . . 77
- 4.6 More on Semilattices . . . 78
- 4.7 Lifting the Typing Framework to err, app, and eff . . . 80

4.8 Kildall's Algorithm . . . 82
4.9 More about Options . . . 85
4.10 The Lightweight Bytecode Verifier . . . 87
4.11 Correctness of the LBV . . . 91
4.12 Completeness of the LBV . . . 93
4.13 Abstract Bytecode Verifier . . . 95
4.14 Semilattices . . . 95
4.15 The Java Type System as Semilattice . . . 95
4.16 The JVM Type System as Semilattice . . . 97
4.17 Effect of Instructions on the State Type . . . 102
4.18 Monotonicity of eff and app . . . 108
4.19 The Bytecode Verifier . . . 109
4.20 The Typing Framework for the JVM . . . 111
4.21 LBV for the JVM . . . 113
4.22 BV Type Safety Invariant . . . 115
4.23 BV Type Safety Proof . . . 120
4.24 Welltyped Programs produce no Type Errors . . . 127
4.25 Kildall for the JVM . . . 130
4.26 Example Welltypings . . . 132
4.27 Alternative definition of well-typing of bytecode, used in compiler type correctness proof . . . 163

Chapter 1

Preface

1.1 Introduction

This document contains the automatically generated listings of the Isabelle sources for μJava. μJava is a reduced model of JavaCard, dedicated to the study of the interaction of the source language, byte code, the byte code verifier and the compiler. In order to make the Isabelle sources more accessible, this introduction provides a brief survey of the main concepts of μJava.

The μJava **source language** (see Chapter 2) only comprises a part of the original JavaCard language. It models features such as:

- The basic "primitive types" of Java
- Object orientation, in particular classes, and relevant relations on classes (subclass, widening)
- Methods and method signatures
- Inheritance and overriding of methods, dynamic binding
- Representatives of "relevant" expressions and statements
- Generation and propagation of system exceptions

However, the following features are missing in μJava wrt. JavaCard:

- Some primitive types (`byte, short`)
- Interfaces and related concepts, arrays
- Most numeric operations, syntactic variants of statements (`do`-loop, `for`-loop)
- Complex block structure, method bodies with multiple returns
- Abrupt termination (`break, continue`)
- Class and method modifiers (such as `static` and `public/private` access modifiers)
- User-defined exception classes and an explicit `throw`-statement. Exceptions cannot be caught.

- A "definite assignment" check

In addition, features are missing that are not part of the JavaCard language, such as multithreading and garbage collection. No attempt has been made to model peculiarities of JavaCard such as the applet firewall or the transaction mechanism.

For a more complete Isabelle model of JavaCard, the reader should consult the Bali formalization (https://isabelle.in.tum.de/verificard/Bali/document.pdf), which models most of the source language features of JavaCard, however without describing the bytecode level.

The central topics of the source language formalization are:

- Description of the structure of the "runtime environment", in particular structure of classes and the program state
- Definition of syntax, typing rules and operational semantics of statements and expressions
- Definition of "conformity" (characterizing type safety) and a type safety proof

The μJava **virtual machine** (see Chapter 3) corresponds rather directly to the source level, in the sense that the same data types are supported and bytecode instructions required for emulating the source level operations are provided. Again, only one representative of different variants of instructions has been selected; for example, there is only one comparison operator. The formalization of the bytecode level is purely descriptive ("no theorems") and rather brief as compared to the source level; all questions related to type systems for and type correctness of bytecode are dealt with in chapter on bytecode verification.

The problem of **bytecode verification** (see Chapter 4) is dealt with in several stages:

- First, the notion of "method type" is introduced, which corresponds to the notion of "type" on the source level.
- Well-typedness of instructions wrt. a method type is defined (see Section 4.19). Roughly speaking, determining well-typedness is *type checking*.
- It is shown that bytecode that is well-typed in this sense can be safely executed – a type soundness proof on the bytecode level (Section 4.23).
- Given raw bytecode, one of the purposes of bytecode verification is to determine a method type that is well-typed according to the above definition. Roughly speaking, this is *type inference*. The Isabelle formalization presents bytecode verification as an instance of an abstract dataflow algorithm (Kildall's algorithm, see Sections 4.8 to 4.25).

Bytecode verification in μJava so far takes into account:

- Operations and branching instructions
- Exceptions

Initialization during object creation is not accounted for in the present document (see the formalization in https://isabelle.in.tum.de/verificard/obj-init/document.pdf), neither is the `jsr` instruction.

1.2 Theory Dependencies

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

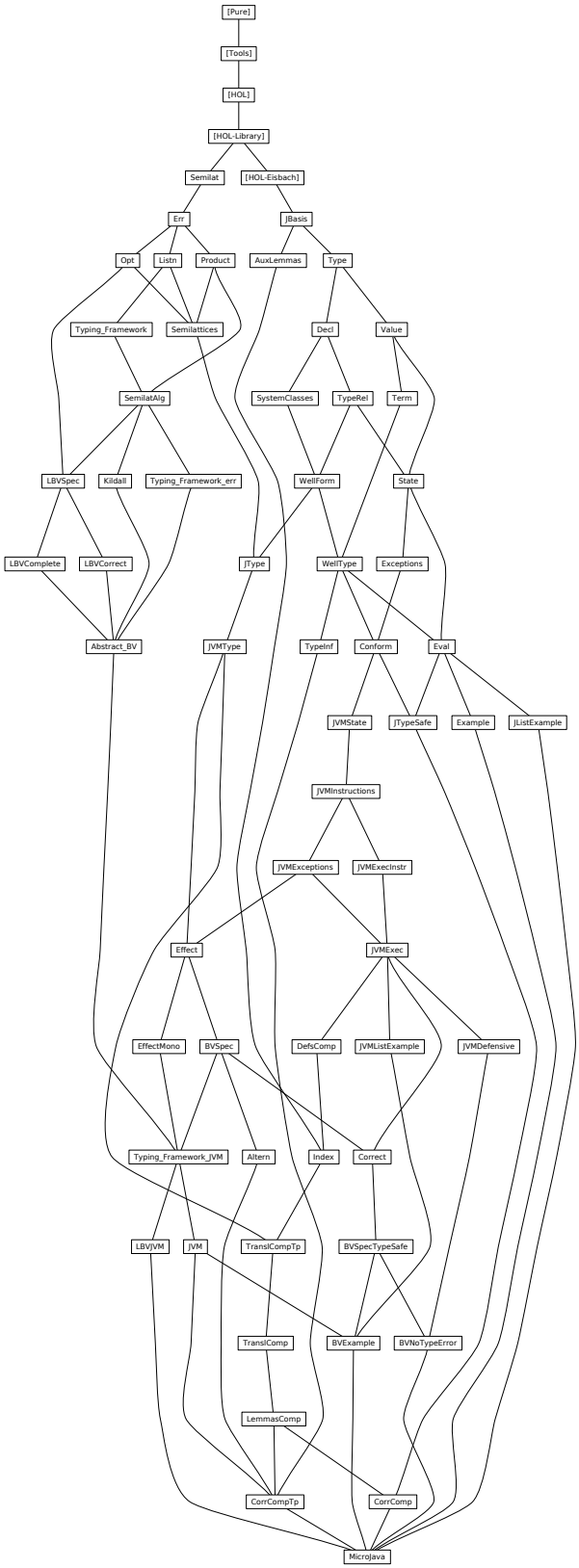


Figure 1.1: Theory Dependency Graph

Chapter 2

Java Source Language

2.1 Some Auxiliary Definitions

theory *JBasis*
imports
  Main
  "HOL-Library.Transitive_Closure_Table"
  "HOL-Eisbach.Eisbach"
begin

lemmas *[simp] = Let_def*

2.1.1 unique

definition *unique :: "('a × 'b) list => bool"* **where**
  "unique == distinct ∘ map fst"

lemma *fst_in_set_lemma: "(x, y) ∈ set xys ⟹ x ∈ fst ‘ set xys"*
  ⟨*proof*⟩

lemma *unique_Nil [simp]: "unique []"*
  ⟨*proof*⟩

lemma *unique_Cons [simp]: "unique ((x,y)#l) = (unique l & (∀y. (x,y) ∉ set l))"*
  ⟨*proof*⟩

lemma *unique_append: "unique l' ⟹ unique l ⟹*
    (∀(x,y) ∈ set l. ∀(x',y') ∈ set l'. x' ≠ x) ⟹ unique (l @ l')"
  ⟨*proof*⟩

lemma *unique_map_inj: "unique l ==> inj f ==> unique (map (%(k,x). (f k, g k x)) l)"*
  ⟨*proof*⟩

2.1.2 More about Maps

lemma *map_of_SomeI: "unique l ⟹ (k, x) ∈ set l ⟹ map_of l k = Some x"*
  ⟨*proof*⟩

lemma *Ball_set_table: "(∀(x,y)∈set l. P x y) ==> (∀x. ∀y. map_of l x = Some y --> P*

```
x y)"
```
⟨*proof*⟩

lemma *table_of_remap_SomeD:*
```
  "map_of (map (λ((k,k'),x). (k,(k',x))) t) k = Some (k',x) ==>
    map_of t (k, k') = Some x"
```
⟨*proof*⟩

end

2.2 Java types

theory *Type* **imports** *JBasis* **begin**

typedecl *cnam*

instantiation *cnam :: equal*
begin

definition *"HOL.equal (cn :: cnam) cn' ⟷ cn = cn'"*
instance ⟨*proof*⟩

end

These instantiations only ensure that the merge in theory *MicroJava* succeeds. FIXME

instantiation *cnam :: typerep*
begin

definition *"typerep_class.typerep ≡ λ_ :: cnam itself. Typerep.Typerep (STR ''Type.cnam'') []"*
instance ⟨*proof*⟩

end

instantiation *cnam :: term_of*
begin

definition *"term_of_class.term_of (C :: cnam) ≡*
```
  Code_Evaluation.Const (STR ''dummy_cnam'') (Typerep.Typerep (STR ''Type.cnam'') [])"
```
instance ⟨*proof*⟩

end

instantiation *cnam :: partial_term_of*
begin

definition *"partial_term_of_class.partial_term_of (C :: cnam itself) n = undefined"*
instance ⟨*proof*⟩

end

— exceptions
datatype

```
  xcpt
  = NullPointer
  | ClassCast
  | OutOfMemory
```

— class names
datatype *cname*
```
  = Object
  | Xcpt xcpt
  | Cname cnam
```

typedecl *vnam* — variable or field name

instantiation *vnam :: equal*
begin

definition *"HOL.equal (vn :: vnam) vn' ⟷ vn = vn'"*
instance ⟨*proof*⟩

end

instantiation *vnam :: typerep*
begin

definition *"typerep_class.typerep ≡ λ_ :: vnam itself. Typerep.Typerep (STR ''Type.vnam'') []"*
instance ⟨*proof*⟩

end

instantiation *vnam :: term_of*
begin

definition *"term_of_class.term_of (V :: vnam) ≡*
Code_Evaluation.Const (STR ''dummy_vnam'') (Typerep.Typerep (STR ''Type.vnam'') [])"
instance ⟨*proof*⟩

end

instantiation *vnam :: partial_term_of*
begin

definition *"partial_term_of_class.partial_term_of (V :: vnam itself) n = undefined"*
instance ⟨*proof*⟩

end

typedecl *mname* — method name

instantiation *mname :: equal*
begin

definition *"HOL.equal (M :: mname) M' ⟷ M = M'"*
instance ⟨*proof*⟩

end

instantiation *mname :: typerep*
begin

definition *"typerep_class.typerep ≡ λ_ :: mname itself. Typerep.Typerep (STR ''Type.mname'') []"*
instance ⟨*proof*⟩

end

instantiation *mname :: term_of*
begin

definition *"term_of_class.term_of (M :: mname) ≡*
Code_Evaluation.Const (STR ''dummy_mname'') (Typerep.Typerep (STR ''Type.mname'') [])"
instance ⟨*proof*⟩

end

instantiation *mname :: partial_term_of*
begin

definition *"partial_term_of_class.partial_term_of (M :: mname itself) n = undefined"*
instance ⟨*proof*⟩

end

— names for *This* pointer and local/field variables
datatype *vname*
= This
| VName vnam

— primitive type, cf. 4.2
datatype *prim_ty*
= Void — 'result type' of void methods
| Boolean
| Integer

— reference type, cf. 4.3
datatype *ref_ty*
= NullT — null type, cf. 4.1
| ClassT cname — class type

— any type, cf. 4.1
datatype *ty*
= PrimT prim_ty — primitive type
| RefT ref_ty — reference type

abbreviation *NT :: ty*
where *"NT == RefT NullT"*

abbreviation *Class :: "cname => ty"*

where *"Class C == RefT (ClassT C)"*

end

2.3 Class Declarations and Programs

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

type_synonym
fdecl = *"vname × ty"* — field declaration, cf. 8.3 (, 9.3)

type_synonym
sig = *"mname × ty list"* — signature of a method, cf. 8.4.2

type_synonym
'c mdecl = *"sig × ty × 'c"* — method declaration in a class

type_synonym
'c "class" = *"cname × fdecl list × 'c mdecl list"*
— class = superclass, fields, methods

type_synonym
'c cdecl = *"cname × 'c class"* — class declaration, cf. 8.1

type_synonym
'c prog = *"'c cdecl list"* — program

translations
(type) "fdecl" <= (type) "vname × ty"
(type) "sig" <= (type) "mname × ty list"
(type) "'c mdecl" <= (type) "sig × ty × 'c"
(type) "'c class" <= (type) "cname × fdecl list × ('c mdecl) list"
(type) "'c cdecl" <= (type) "cname × ('c class)"
(type) "'c prog" <= (type) "('c cdecl) list"

definition *"class" :: "'c prog => (cname ⇀ 'c class)"* **where**
"class ≡ map_of"

definition *is_class :: "'c prog => cname => bool"* **where**
"is_class G C ≡ class G C ≠ None"

lemma *finite_is_class: "finite {C. is_class G C}"*
⟨*proof*⟩

primrec *is_type :: "'c prog => ty => bool"* **where**
"is_type G (PrimT pt) = True"
| "is_type G (RefT t) = (case t of NullT => True | ClassT C => is_class G C)"

end

2.4 Relations between Java Types

theory *TypeRel*
imports *Decl*
begin

— direct subclass, cf. 8.1.3

inductive_set
  subcls1 :: "'c prog => (cname × cname) set"
  and *subcls1' :: "'c prog => cname ⇒ cname => bool" (<_ ⊢ _ ≺C1 _> [71,71,71] 70)*
  for *G :: "'c prog"*
where
  "G ⊢ C ≺C1 D ≡ (C, D) ∈ subcls1 G"
  | subcls1I: "⟦class G C = Some (D,rest); C ≠ Object⟧ ⟹ G ⊢ C ≺C1 D"

abbreviation
  subcls :: "'c prog => cname ⇒ cname => bool" (<_ ⊢ _ ⪯C _> [71,71,71] 70)
  where *"G ⊢ C ⪯C D ≡ (C, D) ∈ (subcls1 G)*"*

lemma *subcls1D:*
  "G⊢C≺C1D ⟹ C ≠ Object ∧ (∃fs ms. class G C = Some (D,fs,ms))"
⟨*proof*⟩

lemma *subcls1_def2:*
  "subcls1 P =
     (SIGMA C:{C. is_class P C}. {D. C≠Object ∧ fst (the (class P C))=D})"
  ⟨*proof*⟩

lemma *finite_subcls1: "finite (subcls1 G)"*
⟨*proof*⟩

lemma *subcls_is_class: "(C, D) ∈ (subcls1 G)$^+$ ⟹ is_class G C"*
⟨*proof*⟩

lemma *subcls_is_class2 [rule_format (no_asm)]:*
  "G⊢C⪯C D ⟹ is_class G D ⟶ is_class G C"
⟨*proof*⟩

definition *class_rec :: "'c prog ⇒ cname ⇒ 'a ⇒*
    (cname ⇒ fdecl list ⇒ 'c mdecl list ⇒ 'a ⇒ 'a) ⇒ 'a" **where**
  "class_rec G == wfrec ((subcls1 G)$^{-1}$)
    (λr C t f. case class G C of
         None ⇒ undefined
       | Some (D,fs,ms) ⇒
          f C fs ms (if C = Object then t else r D t f))"

lemma *class_rec_lemma:*
  assumes *wf: "wf ((subcls1 G)$^{-1}$)"*
    and *cls: "class G C = Some (D, fs, ms)"*
  shows *"class_rec G C t f = f C fs ms (if C=Object then t else class_rec G D t f)"*
  ⟨*proof*⟩

definition

```
"wf_class G = wf ((subcls1 G)^-1)"
```

Code generator setup

code_pred
```
(modes: i ⇒ i ⇒ o ⇒ bool, i ⇒ i ⇒ i ⇒ bool)
subcls1p
```
⟨*proof*⟩

declare *subcls1_def [code_pred_def]*

code_pred
```
(modes: i ⇒ i × o ⇒ bool, i ⇒ i × i ⇒ bool)
[inductify]
subcls1
```
⟨*proof*⟩

definition *subcls'* **where** *"subcls' G = (subcls1p G)^**"*

code_pred
```
(modes: i ⇒ i ⇒ i ⇒ bool, i ⇒ i ⇒ o ⇒ bool)
[inductify]
subcls'
```
⟨*proof*⟩

lemma *subcls_conv_subcls' [code_unfold]:*
```
"(subcls1 G)* = {(C, D). subcls' G C D}"
```
⟨*proof*⟩

lemma *class_rec_code [code]:*
```
"class_rec G C t f =
(if wf_class G then
  (case class G C of
     None ⇒ class_rec G C t f
   | Some (D, fs, ms) ⇒
     if C = Object then f Object fs ms t else f C fs ms (class_rec G D t f))
 else class_rec G C t f)"
```
⟨*proof*⟩

lemma *wf_class_code [code]:*
```
"wf_class G ⟷ (∀(C, rest) ∈ set G. C ≠ Object ⟶ ¬ G ⊢ fst (the (class G C)) ≺C
C)"
```
⟨*proof*⟩

definition *"method" :: "'c prog × cname => (sig ⇀ cname × ty × 'c)"*
— methods of a class, with inheritance, overriding and hiding, cf. 8.4.6
where *[code]: "method ≡ λ(G,C). class_rec G C Map.empty (λC fs ms ts.*
ts ++ map_of (map (λ(s,m). (s,(C,m))) ms))"

definition *fields :: "'c prog × cname => ((vname × cname) × ty) list"*
— list of fields of a class, including inherited and hidden ones
where *[code]: "fields ≡ λ(G,C). class_rec G C [] (λC fs ms ts.*
map (λ(fn,ft). ((fn,C),ft)) fs @ ts)"

definition *field :: "'c prog × cname => (vname ⇀ cname × ty)"*

where *[code]: "field == map_of o (map (λ((fn,fd),ft). (fn,(fd,ft)))) o fields"*

lemma *method_rec_lemma: "[|class G C = Some (D,fs,ms); wf ((subcls1 G)⁻¹)|] ==>*
method (G,C) = (if C = Object then Map.empty else method (G,D)) ++
map_of (map (λ(s,m). (s,(C,m))) ms)"
⟨*proof*⟩

lemma *fields_rec_lemma: "[|class G C = Some (D,fs,ms); wf ((subcls1 G)⁻¹)|] ==>*
fields (G,C) =
map (λ(fn,ft). ((fn,C),ft)) fs @ (if C = Object then [] else fields (G,D))"
⟨*proof*⟩

lemma *field_fields:*
"field (G,C) fn = Some (fd, fT) ⟹ map_of (fields (G,C)) (fn, fd) = Some fT"
⟨*proof*⟩
inductive
widen :: "'c prog => [ty , ty] => bool" (<_ ⊢ _ ≼ _> [71,71,71] 70)
for *G :: "'c prog"*
where
refl [intro!, simp]: "G⊢ T ≼ T" — identity conv., cf. 5.1.1
| *subcls : "G⊢C≼C D ==> G⊢Class C ≼ Class D"*
| *null [intro!]: "G⊢ NT ≼ RefT R"*

code_pred *widen* ⟨*proof*⟩

lemmas *refl = HOL.refl*

— casting conversion, cf. 5.5 / 5.1.5
— left out casts on primitve types
inductive
cast :: "'c prog => [ty , ty] => bool" (<_ ⊢ _ ≼? _> [71,71,71] 70)
for *G :: "'c prog"*
where
widen: "G⊢ C≼ D ==> G⊢C ≼? D"
| *subcls: "G⊢ D≼C C ==> G⊢Class C ≼? Class D"*

lemma *widen_PrimT_RefT [iff]: "(G⊢PrimT pT≼RefT rT) = False"*
⟨*proof*⟩

lemma *widen_RefT: "G⊢RefT R≼T ==> ∃t. T=RefT t"*
⟨*proof*⟩

lemma *widen_RefT2: "G⊢S≼RefT R ==> ∃t. S=RefT t"*
⟨*proof*⟩

lemma *widen_Class: "G⊢Class C≼T ==> ∃D. T=Class D"*
⟨*proof*⟩

lemma *widen_Class_NullT [iff]: "(G⊢Class C≼NT) = False"*
⟨*proof*⟩

lemma *widen_Class_Class [iff]: "(G⊢Class C≼ Class D) = (G⊢C≼C D)"*
⟨*proof*⟩

lemma *widen_NT_Class [simp]: "G ⊢ T ≼ NT ⟹ G ⊢ T ≼ Class D"*
⟨*proof*⟩

lemma *cast_PrimT_RefT [iff]: "(G⊢PrimT pT≼? RefT rT) = False"*
⟨*proof*⟩

lemma *cast_RefT: "G ⊢ C ≼? Class D ⟹ ∃ rT. C = RefT rT"*
⟨*proof*⟩

theorem *widen_trans[trans]: "⟦G⊢S≼U; G⊢U≼T⟧ ⟹ G⊢S≼T"*
⟨*proof*⟩

end

2.5 Java Values

theory *Value* **imports** *Type* **begin**

typedecl *loc'* — locations, i.e. abstract references on objects

datatype *loc*
  = XcptRef xcpt — special locations for pre-allocated system exceptions
  | Loc loc' — usual locations (references on objects)

datatype *val*
  = Unit — dummy result value of void methods
  | Null — null reference
  | Bool bool — Boolean value
  | Intg int — integer value, name Intg instead of Int because of clash with HOL/Set.thy
  | Addr loc — addresses, i.e. locations of objects

primrec *the_Bool :: "val => bool"* **where**
  "the_Bool (Bool b) = b"

primrec *the_Intg :: "val => int"* **where**
  "the_Intg (Intg i) = i"

primrec *the_Addr :: "val => loc"* **where**
  "the_Addr (Addr a) = a"

primrec *defpval :: "prim_ty => val"* — default value for primitive types **where**
  "defpval Void   = Unit"
| "defpval Boolean = Bool False"
| "defpval Integer = Intg 0"

primrec *default_val :: "ty => val"* — default value for all types **where**
  "default_val (PrimT pt) = defpval pt"
| "default_val (RefT r) = Null"

end

2.6 Program State

theory *State*
imports *TypeRel Value*
begin

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_vars :: "('a × ty) list => ('a ⇀ val)"* **where**
"init_vars == map_of o map (λ(n,T). (n,default_val T))"

type_synonym *aheap = "loc ⇀ obj"* — "*heap*" used in a translation below
type_synonym *locals = "vname ⇀ val"* — simple state, i.e. variable contents

type_synonym *state = "aheap × locals"* — heap, local parameter including This
type_synonym *xstate = "val option × state"* — state including exception information

abbreviation *(input)*
heap :: "state => aheap"
where *"heap == fst"*

abbreviation *(input)*
locals :: "state => locals"
where *"locals == snd"*

abbreviation *"Norm s == (None, s)"*

abbreviation *(input)*
abrupt :: "xstate ⇒ val option"
where *"abrupt == fst"*

abbreviation *(input)*
store :: "xstate ⇒ state"
where *"store == snd"*

abbreviation
lookup_obj :: "state ⇒ val ⇒ obj"
where *"lookup_obj s a' == the (heap s (the_Addr a'))"*

definition *raise_if :: "bool ⇒ xcpt ⇒ val option ⇒ val option"* **where**
"raise_if b x xo ≡ if b ∧ (xo = None) then Some (Addr (XcptRef x)) else xo"

Make *new_Addr* completely specified (at least for the code generator)

consts *nat_to_loc' :: "nat => loc'"*
code_datatype *nat_to_loc'*
definition *new_Addr :: "aheap => loc × val option"* **where**
"new_Addr h ≡

```
  if ∃n. h (Loc (nat_to_loc' n)) = None
  then (Loc (nat_to_loc' (LEAST n. h (Loc (nat_to_loc' n)) = None)), None)
  else (Loc (nat_to_loc' 0), Some (Addr (XcptRef OutOfMemory)))"
```

definition *np :: "val => val option => val option"* **where**
"np v == raise_if (v = Null) NullPointer"

definition *c_hupd :: "aheap => xstate => xstate"* **where**
"c_hupd h'== λ(xo,(h,l)). if xo = None then (None,(h',l)) else (xo,(h,l))"

definition *cast_ok :: "'c prog => cname => aheap => val => bool"* **where**
"cast_ok G C h v == v = Null ∨ G⊢obj_ty (the (h (the_Addr v)))≼ Class C"

lemma *obj_ty_def2 [simp]: "obj_ty (C,fs) = Class C"*
⟨*proof*⟩

lemma *new_AddrD: "new_Addr hp = (ref, xcp) ⟹*
hp ref = None ∧ xcp = None ∨ xcp = Some (Addr (XcptRef OutOfMemory))"
⟨*proof*⟩

lemma *raise_if_True [simp]: "raise_if True x y ≠ None"*
⟨*proof*⟩

lemma *raise_if_False [simp]: "raise_if False x y = y"*
⟨*proof*⟩

lemma *raise_if_Some [simp]: "raise_if c x (Some y) ≠ None"*
⟨*proof*⟩

lemma *raise_if_Some2 [simp]:*
"raise_if c z (if x = None then Some y else x) ≠ None"
⟨*proof*⟩

lemma *raise_if_SomeD [rule_format (no_asm)]:*
"raise_if c x y = Some z ⟶ c ∧ Some z = Some (Addr (XcptRef x)) | y = Some z"
⟨*proof*⟩

lemma *raise_if_NoneD [rule_format (no_asm)]:*
"raise_if c x y = None --> ¬ c ∧ y = None"
⟨*proof*⟩

lemma *np_NoneD [rule_format (no_asm)]:*
"np a' x' = None --> x' = None ∧ a' ≠ Null"
⟨*proof*⟩

lemma *np_None [rule_format (no_asm), simp]: "a' ≠ Null --> np a' x' = x'"*
⟨*proof*⟩

lemma *np_Some [simp]: "np a' (Some xc) = Some xc"*
⟨*proof*⟩

lemma *np_Null [simp]: "np Null None = Some (Addr (XcptRef NullPointer))"*
⟨*proof*⟩

lemma *np_Addr [simp]: "np (Addr a) None = None"*
⟨*proof*⟩

lemma *np_raise_if [simp]: "(np Null (raise_if c xc None)) =*
Some (Addr (XcptRef (if c then xc else NullPointer)))"
⟨*proof*⟩

lemma *c_hupd_fst [simp]: "fst (c_hupd h (x, s)) = x"*
⟨*proof*⟩

Naive implementation for *new_Addr* by exhaustive search

definition *gen_new_Addr :: "aheap => nat ⇒ loc × val option"* **where**
"gen_new_Addr h n ≡
if ∃a. a ≥ n ∧ h (Loc (nat_to_loc' a)) = None
then (Loc (nat_to_loc' (LEAST a. a ≥ n ∧ h (Loc (nat_to_loc' a)) = None)), None)
else (Loc (nat_to_loc' 0), Some (Addr (XcptRef OutOfMemory)))"

lemma *new_Addr_code_code [code]:*
"new_Addr h = gen_new_Addr h 0"
⟨*proof*⟩

lemma *gen_new_Addr_code [code]:*
"gen_new_Addr h n = (if h (Loc (nat_to_loc' n)) = None then (Loc (nat_to_loc' n), None)
else gen_new_Addr h (Suc n))"
⟨*proof*⟩

instantiation *loc' :: equal*
begin

definition *"HOL.equal (l :: loc') l' ⟷ l = l'"*
instance ⟨*proof*⟩

end

end

2.7 Expressions and Statements

theory *Term* **imports** *Value* **begin**

datatype *binop = Eq | Add* — function codes for binary operation

datatype *expr*
= NewC cname — class instance creation
| Cast cname expr — type cast
| Lit val — literal value, also references
| BinOp binop expr expr — binary operation
| LAcc vname — local (incl. parameter) access
| LAss vname expr *(<_::=_> [90,90]90)* — local assign
| FAcc cname expr vname *(<{_}_.._> [10,90,99]90)* — field access
| FAss cname expr vname expr *(<{_}_.._:=_> [10,90,99,90]90)* — field ass.

```
  | Call cname expr mname
    "ty list" "expr list"    (<{_}_.._'( {_}_')> [10,90,99,10,10] 90)
```
— method call

datatype_compat *expr*

datatype *stmt*
```
  = Skip                        — empty statement
  | Expr expr                   — expression statement
  | Comp stmt stmt       (<_;; _>              [61,60]60)
  | Cond expr stmt stmt  (<If '(_') _ Else _>  [80,79,79]70)
  | Loop expr stmt       (<While '(_') _>      [80,79]70)
```

end

2.8 System Classes

theory *SystemClasses* **imports** *Decl* **begin**

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

definition *ObjectC :: "'c cdecl"* **where**
```
  [code_unfold]: "ObjectC ≡ (Object, (undefined,[],[]))"
```

definition *NullPointerC :: "'c cdecl"* **where**
```
  [code_unfold]: "NullPointerC ≡ (Xcpt NullPointer, (Object,[],[]))"
```

definition *ClassCastC :: "'c cdecl"* **where**
```
  [code_unfold]: "ClassCastC ≡ (Xcpt ClassCast, (Object,[],[]))"
```

definition *OutOfMemoryC :: "'c cdecl"* **where**
```
  [code_unfold]: "OutOfMemoryC ≡ (Xcpt OutOfMemory, (Object,[],[]))"
```

definition *SystemClasses :: "'c cdecl list"* **where**
```
  [code_unfold]: "SystemClasses ≡ [ObjectC, NullPointerC, ClassCastC, OutOfMemoryC]"
```

end

2.9 Well-formedness of Java programs

theory *WellForm*
imports *TypeRel SystemClasses*
begin

for static checks on expressions and statements, see WellType.

improvements over Java Specification 1.0 (cf. 8.4.6.3, 8.4.6.4, 9.4.1):

- a method implementing or overwriting another method may have a result type that widens to the result type of the other method (instead of identical type)

simplifications:

- for uniformity, Object is assumed to be declared like any other class

type_synonym *'c wf_mb = "'c prog => cname => 'c mdecl => bool"*

definition *wf_syscls :: "'c prog => bool"* **where**
"wf_syscls G == let cs = set G in Object ∈ fst ‘ cs ∧ (∀x. Xcpt x ∈ fst ‘ cs)"

definition *wf_fdecl :: "'c prog => fdecl => bool"* **where**
"wf_fdecl G == λ(fn,ft). is_type G ft"

definition *wf_mhead :: "'c prog => sig => ty => bool"* **where**
"wf_mhead G == λ(mn,pTs) rT. (∀T∈set pTs. is_type G T) ∧ is_type G rT"

definition *ws_cdecl :: "'c prog => 'c cdecl => bool"* **where**
"ws_cdecl G ==
λ(C,(D,fs,ms)).
(∀f∈set fs. wf_fdecl G f) ∧ unique fs ∧
(∀(sig,rT,mb)∈set ms. wf_mhead G sig rT) ∧ unique ms ∧
(C ≠ Object ⟶ is_class G D ∧ ¬G⊢D≼C C)"

definition *ws_prog :: "'c prog => bool"* **where**
"ws_prog G ==
wf_syscls G ∧ (∀c∈set G. ws_cdecl G c) ∧ unique G"

definition *wf_mrT :: "'c prog => 'c cdecl => bool"* **where**
"wf_mrT G ==
λ(C,(D,fs,ms)).
(C ≠ Object ⟶ (∀(sig,rT,b)∈set ms. ∀D' rT' b'.
method(G,D) sig = Some(D',rT',b') --> G⊢rT≼rT'))"

definition *wf_cdecl_mdecl :: "'c wf_mb => 'c prog => 'c cdecl => bool"* **where**
"wf_cdecl_mdecl wf_mb G ==
λ(C,(D,fs,ms)). (∀m∈set ms. wf_mb G C m)"

definition *wf_prog :: "'c wf_mb => 'c prog => bool"* **where**
"wf_prog wf_mb G ==
ws_prog G ∧ (∀c∈ set G. wf_mrT G c ∧ wf_cdecl_mdecl wf_mb G c)"

definition *wf_mdecl :: "'c wf_mb => 'c wf_mb"* **where**
"wf_mdecl wf_mb G C == λ(sig,rT,mb). wf_mhead G sig rT ∧ wf_mb G C (sig,rT,mb)"

definition *wf_cdecl :: "'c wf_mb => 'c prog => 'c cdecl => bool"* **where**
"wf_cdecl wf_mb G ==
λ(C,(D,fs,ms)).
(∀f∈set fs. wf_fdecl G f) ∧ unique fs ∧
(∀m∈set ms. wf_mdecl wf_mb G C m) ∧ unique ms ∧
(C ≠ Object ⟶ is_class G D ∧ ¬G⊢D≼C C ∧
(∀(sig,rT,b)∈set ms. ∀D' rT' b'.
method(G,D) sig = Some(D',rT',b') --> G⊢rT≼rT'))"

lemma *wf_cdecl_mrT_cdecl_mdecl:*
"(wf_cdecl wf_mb G c) = (ws_cdecl G c ∧ wf_mrT G c ∧ wf_cdecl_mdecl wf_mb G c)"
⟨*proof*⟩

lemma *wf_cdecl_ws_cdecl [intro]: "wf_cdecl wf_mb G cd ⟹ ws_cdecl G cd"*
⟨*proof*⟩

lemma *wf_prog_ws_prog [intro]: "wf_prog wf_mb G ⟹ ws_prog G"*
⟨*proof*⟩

lemma *wf_prog_wf_mdecl:*
"⟦ wf_prog wf_mb G; (C,S,fs,mdecls) ∈ set G; ((mn,pTs),rT,code) ∈ set mdecls⟧
⟹ wf_mdecl wf_mb G C ((mn,pTs),rT,code)"
⟨*proof*⟩

lemma *class_wf:*
"[|class G C = Some c; wf_prog wf_mb G|]
==> wf_cdecl wf_mb G (C,c) ∧ wf_mrT G (C,c)"
⟨*proof*⟩

lemma *class_wf_struct:*
"[|class G C = Some c; ws_prog G|]
==> ws_cdecl G (C,c)"
⟨*proof*⟩

lemma *class_Object [simp]:*
"ws_prog G ==> ∃X fs ms. class G Object = Some (X,fs,ms)"
⟨*proof*⟩

lemma *class_Object_syscls [simp]:*
"wf_syscls G ==> unique G ⟹ ∃X fs ms. class G Object = Some (X,fs,ms)"
⟨*proof*⟩

lemma *is_class_Object [simp]: "ws_prog G ==> is_class G Object"*
⟨*proof*⟩

lemma *is_class_xcpt [simp]: "ws_prog G ⟹ is_class G (Xcpt x)"*
⟨*proof*⟩

lemma *subcls1_wfD: "[|G⊢C≺C1D; ws_prog G|] ==> D ≠ C ∧ (D, C) ∉ (subcls1 G)⁺"*
⟨*proof*⟩

lemma *wf_cdecl_supD:*
"!!r. ⟦ws_cdecl G (C,D,r); C ≠ Object⟧ ⟹ is_class G D"
⟨*proof*⟩

lemma *subcls_asym: "[|ws_prog G; (C, D) ∈ (subcls1 G)⁺|] ==> (D, C) ∉ (subcls1 G)⁺"*
⟨*proof*⟩

lemma *subcls_irrefl: "[|ws_prog G; (C, D) ∈ (subcls1 G)⁺|] ==> C ≠ D"*
⟨*proof*⟩

lemma *acyclic_subcls1: "ws_prog G ==> acyclic (subcls1 G)"*
⟨*proof*⟩

lemma *wf_subcls1: "ws_prog G ==> wf ((subcls1 G)⁻¹)"*
⟨*proof*⟩

lemma *subcls_induct_struct:*
```
"[|ws_prog G; !!C. ∀D. (C, D) ∈ (subcls1 G)^+ --> P D ==> P C|] ==> P C"
(is "?A ⟹ PROP ?P ⟹ _")
```
⟨*proof*⟩

lemma *subcls_induct:*
```
"[|wf_prog wf_mb G; !!C. ∀D. (C, D) ∈ (subcls1 G)^+ --> P D ==> P C|] ==> P C"
(is "?A ⟹ PROP ?P ⟹ _")
```
⟨*proof*⟩

lemma *subcls1_induct:*
```
"[|is_class G C; wf_prog wf_mb G; P Object;
   !!C D fs ms. [|C ≠ Object; is_class G C; class G C = Some (D,fs,ms) ∧
    wf_cdecl wf_mb G (C,D,fs,ms) ∧ G⊢C≺C1D ∧ is_class G D ∧ P D|] ==> P C
 |] ==> P C"
(is "?A ⟹ ?B ⟹ ?C ⟹ PROP ?P ⟹ _")
```
⟨*proof*⟩

lemma *subcls1_induct_struct:*
```
"[|is_class G C; ws_prog G; P Object;
   !!C D fs ms. [|C ≠ Object; is_class G C; class G C = Some (D,fs,ms) ∧
    ws_cdecl G (C,D,fs,ms) ∧ G⊢C≺C1D ∧ is_class G D ∧ P D|] ==> P C
 |] ==> P C"
(is "?A ⟹ ?B ⟹ ?C ⟹ PROP ?P ⟹ _")
```
⟨*proof*⟩

lemmas *method_rec = wf_subcls1 [THEN [2] method_rec_lemma]*

lemmas *fields_rec = wf_subcls1 [THEN [2] fields_rec_lemma]*

lemma *field_rec:*
```
"⟦class G C = Some (D, fs, ms); ws_prog G⟧
⟹ field (G, C) =
   (if C = Object then Map.empty else field (G, D)) ++
   map_of (map (λ(s, f). (s, C, f)) fs)"
```
⟨*proof*⟩

lemma *method_Object [simp]:*
```
"method (G, Object) sig = Some (D, mh, code) ⟹ ws_prog G ⟹ D = Object"
```
⟨*proof*⟩

lemma *fields_Object [simp]:*
```
"⟦ ((vn, C), T) ∈ set (fields (G, Object)); ws_prog G ⟧
  ⟹ C = Object"
```
⟨*proof*⟩

lemma *subcls_C_Object:*
```
"[|is_class G C; ws_prog G|] ==> G⊢C≼C Object"
```
⟨*proof*⟩

lemma *is_type_rTI:*
```
"wf_mhead G sig rT ==> is_type G rT"
```
⟨*proof*⟩

lemma *widen_fields_defpl':*
```
"[|is_class G C; ws_prog G|] ==>
  ∀((fn,fd),fT)∈set (fields (G,C)). G⊢C≼C fd"
```
⟨*proof*⟩

lemma *widen_fields_defpl:*
```
"[|((fn,fd),fT) ∈ set (fields (G,C)); ws_prog G; is_class G C|] ==>
G⊢C≼C fd"
```
⟨*proof*⟩

lemma *unique_fields:*
```
"[|is_class G C; ws_prog G|] ==> unique (fields (G,C))"
```
⟨*proof*⟩

lemma *fields_mono_lemma [rule_format (no_asm)]:*
```
"[|ws_prog G; (C', C) ∈ (subcls1 G)* |] ==>
x ∈ set (fields (G,C)) --> x ∈ set (fields (G,C'))"
```
⟨*proof*⟩

lemma *fields_mono:*
```
"⟦map_of (fields (G,C)) fn = Some f; G⊢D≼C C; is_class G D; ws_prog G⟧
  ⟹ map_of (fields (G,D)) fn = Some f"
```
⟨*proof*⟩

lemma *widen_cfs_fields:*
```
"[|field (G,C) fn = Some (fd, fT); G⊢D≼C C; ws_prog G|]==>
  map_of (fields (G,D)) (fn, fd) = Some fT"
```
⟨*proof*⟩

lemma *method_wf_mdecl [rule_format (no_asm)]:*
```
"wf_prog wf_mb G ==> is_class G C ⟹
   method (G,C) sig = Some (md,mh,m)
 --> G⊢C≼C md ∧ wf_mdecl wf_mb G md (sig,(mh,m))"
```
⟨*proof*⟩

lemma *method_wf_mhead [rule_format (no_asm)]:*
```
"ws_prog G ==> is_class G C ⟹
   method (G,C) sig = Some (md,rT,mb)
 --> G⊢C≼C md ∧ wf_mhead G sig rT"
```
⟨*proof*⟩

lemma *subcls_widen_methd [rule_format (no_asm)]:*
```
"[|G⊢T'≼C T; wf_prog wf_mb G|] ==>
 ∀D rT b. method (G,T) sig = Some (D,rT ,b) -->
(∃D' rT' b'. method (G,T') sig = Some (D',rT',b') ∧ G⊢D'≼C D ∧ G⊢rT'≼rT)"
```
⟨*proof*⟩

lemma *subtype_widen_methd:*
```
"[| G⊢ C≼C D; wf_prog wf_mb G;
   method (G,D) sig = Some (md, rT, b) |]
==> ∃mD' rT' b'. method (G,C) sig= Some(mD',rT',b') ∧ G⊢rT'≼rT"
```
⟨*proof*⟩

lemma *method_in_md [rule_format (no_asm)]:*
```
"ws_prog G ==> is_class G C ⟹ ∀D. method (G,C) sig = Some(D,mh,code)
```

```
  --> is_class G D ∧ method (G,D) sig = Some(D,mh,code)"
```
⟨*proof*⟩

lemma *method_in_md_struct [rule_format (no_asm)]:*
```
  "ws_prog G ==> is_class G C ⟹ ∀D. method (G,C) sig = Some(D,mh,code)
  --> is_class G D ∧ method (G,D) sig = Some(D,mh,code)"
```
⟨*proof*⟩

lemma *fields_in_fd [rule_format (no_asm)]: "⟦ wf_prog wf_mb G; is_class G C⟧*
```
  ⟹ ∀ vn D T. (((vn,D),T) ∈ set (fields (G,C))
  ⟶ (is_class G D ∧ ((vn,D),T) ∈ set (fields (G,D))))"
```
⟨*proof*⟩

lemma *field_in_fd [rule_format (no_asm)]: "⟦ wf_prog wf_mb G; is_class G C⟧*
```
  ⟹ ∀ vn D T. (field (G,C) vn = Some(D,T)
  ⟶ is_class G D ∧ field (G,D) vn = Some(D,T))"
```
⟨*proof*⟩

lemma *widen_methd:*
```
"[| method (G,C) sig = Some (md,rT,b); wf_prog wf_mb G; G⊢T''⪯C C|]
  ==> ∃md' rT' b'. method (G,T'') sig = Some (md',rT',b') ∧ G⊢rT'⪯rT"
```
⟨*proof*⟩

lemma *widen_field: "⟦ (field (G,C) fn) = Some (fd, fT); wf_prog wf_mb G; is_class G C*
⟧
```
  ⟹ G⊢C⪯C fd"
```
⟨*proof*⟩

lemma *Call_lemma:*
```
"[|method (G,C) sig = Some (md,rT,b); G⊢T''⪯C C; wf_prog wf_mb G;
  class G C = Some y|] ==> ∃T' rT' b. method (G,T'') sig = Some (T',rT',b) ∧
  G⊢rT'⪯rT ∧ G⊢T''⪯C T' ∧ wf_mhead G sig rT' ∧ wf_mb G T' (sig,rT',b)"
```
⟨*proof*⟩

lemma *fields_is_type_lemma [rule_format (no_asm)]:*
```
  "[|is_class G C; ws_prog G|] ==>
  ∀f∈set (fields (G,C)). is_type G (snd f)"
```
⟨*proof*⟩

lemma *fields_is_type:*
```
  "[|map_of (fields (G,C)) fn = Some f; ws_prog G; is_class G C|] ==>
  is_type G f"
```
⟨*proof*⟩

lemma *field_is_type: "⟦ ws_prog G; is_class G C; field (G, C) fn = Some (fd, fT) ⟧*
```
  ⟹ is_type G fT"
```
⟨*proof*⟩

lemma *methd:*

```
  "[| ws_prog G; (C,S,fs,mdecls) ∈ set G; (sig,rT,code) ∈ set mdecls |]
  ==> method (G,C) sig = Some(C,rT,code) ∧ is_class G C"
```
⟨*proof*⟩

lemma *wf_mb'E:*
```
  "⟦ wf_prog wf_mb G; ⋀C S fs ms m.⟦(C,S,fs,ms) ∈ set G; m ∈ set ms⟧ ⟹ wf_mb' G C m
⟧
  ⟹ wf_prog wf_mb' G"
```
⟨*proof*⟩

lemma *fst_mono:* `"A ⊆ B ⟹ fst ` A ⊆ fst ` B"` ⟨*proof*⟩

lemma *wf_syscls:*
```
  "set SystemClasses ⊆ set G ⟹ wf_syscls G"
```
⟨*proof*⟩

end

2.10 Well-typedness Constraints

theory *WellType* **imports** *Term WellForm* **begin**

the formulation of well-typedness of method calls given below (as well as the Java Specification 1.0) is a little too restrictive: Is does not allow methods of class Object to be called upon references of interface type.

simplifications:

- the type rules include all static checks on expressions and statements, e.g. definedness of names (of parameters, locals, fields, methods)

local variables, including method parameters and This:

type_synonym *lenv = "vname ⇀ ty"*
type_synonym *'c env = "'c prog × lenv"*

abbreviation *(input)*
```
  prg :: "'c env => 'c prog"
```
  where *"prg == fst"*

abbreviation *(input)*
```
  localT :: "'c env => (vname ⇀ ty)"
```
  where *"localT == snd"*

definition *more_spec :: "'c prog ⇒ (ty × 'x) × ty list ⇒ (ty × 'x) × ty list ⇒ bool"*
  where *"more_spec G == λ((d,h),pTs). λ((d',h'),pTs'). G⊢d⪯d' ∧*
list_all2 (λT T'. G⊢T⪯T') pTs pTs'"

definition *appl_methds :: "'c prog ⇒ cname ⇒ sig ⇒ ((ty × ty) × ty list) set"*
  — applicable methods, cf. 15.11.2.1

where *"appl_methds G C == λ(mn, pTs).*
{((Class md,rT),pTs') |md rT mb pTs'.
method (G,C) (mn, pTs') = Some (md,rT,mb) ∧
list_all2 (λT T'. G⊢T⪯T') pTs pTs'}"

definition *max_spec :: "'c prog ⇒ cname ⇒ sig ⇒ ((ty × ty) × ty list) set"*
— maximally specific methods, cf. 15.11.2.2
where *"max_spec G C sig == {m. m ∈appl_methds G C sig ∧*
(∀m'∈appl_methds G C sig.
more_spec G m' m --> m' = m)}"

lemma *max_spec2appl_meths:*
"x ∈ max_spec G C sig ==> x ∈ appl_methds G C sig"
⟨*proof*⟩

lemma *appl_methsD:*
"((md,rT),pTs')∈appl_methds G C (mn, pTs) ==>
∃D b. md = Class D ∧ method (G,C) (mn, pTs') = Some (D,rT,b)
∧ list_all2 (λT T'. G⊢T⪯T') pTs pTs'"
⟨*proof*⟩

lemmas *max_spec2mheads = insertI1 [THEN [2] equalityD2 [THEN subsetD],*
THEN max_spec2appl_meths, THEN appl_methsD]

primrec *typeof :: "(loc => ty option) => val => ty option"*
where
"typeof dt Unit = Some (PrimT Void)"
| "typeof dt Null = Some NT"
| "typeof dt (Bool b) = Some (PrimT Boolean)"
| "typeof dt (Intg i) = Some (PrimT Integer)"
| "typeof dt (Addr a) = dt a"

lemma *is_type_typeof [rule_format (no_asm), simp]:*
"(∀a. v ≠ Addr a) --> (∃T. typeof t v = Some T ∧ is_type G T)"
⟨*proof*⟩

lemma *typeof_empty_is_type [rule_format (no_asm)]:*
"typeof (λa. None) v = Some T ⟶ is_type G T"
⟨*proof*⟩

lemma *typeof_default_val: "∃T. (typeof dt (default_val ty) = Some T) ∧ G⊢ T ⪯ ty"*
⟨*proof*⟩

type_synonym
java_mb = "vname list × (vname × ty) list × stmt × expr"
— method body with parameter names, local variables, block, result expression.
— local variables might include This, which is hidden anyway

inductive
ty_expr :: "'c env => expr => ty => bool" (<_ ⊢ _ :: _> [51, 51, 51] 50)
and *ty_exprs :: "'c env => expr list => ty list => bool" (<_ ⊢ _ [::] _> [51, 51, 51] 50)*
and *wt_stmt :: "'c env => stmt => bool" (<_ ⊢ _ √> [51, 51] 50)*

where

```
  NewC: "[| is_class (prg E) C |] ==>
         E⊢NewC C::Class C"  — cf. 15.8

  — cf. 15.15
| Cast: "[| E⊢e::C; is_class (prg E) D;
            prg E⊢C≼? Class D |] ==>
         E⊢Cast D e:: Class D"

  — cf. 15.7.1
| Lit:    "[| typeof (λv. None) x = Some T |] ==>
         E⊢Lit x::T"


  — cf. 15.13.1
| LAcc: "[| localT E v = Some T; is_type (prg E) T |] ==>
         E⊢LAcc v::T"

| BinOp:"[| E⊢e1::T;
            E⊢e2::T;
            if bop = Eq then T' = PrimT Boolean
                        else T' = T ∧ T = PrimT Integer|] ==>
            E⊢BinOp bop e1 e2::T'"

  — cf. 15.25, 15.25.1
| LAss: "[| v ~= This;
            E⊢LAcc v::T;
            E⊢e::T';
            prg E⊢T'≼T |] ==>
         E⊢v::=e::T'"

  — cf. 15.10.1
| FAcc: "[| E⊢a::Class C;
            field (prg E,C) fn = Some (fd,fT) |] ==>
            E⊢{fd}a..fn::fT"

  — cf. 15.25, 15.25.1
| FAss: "[| E⊢{fd}a..fn::T;
            E⊢v        ::T';
            prg E⊢T'≼T |] ==>
         E⊢{fd}a..fn:=v::T'"


  — cf. 15.11.1, 15.11.2, 15.11.3
| Call: "[| E⊢a::Class C;
            E⊢ps[::]pTs;
            max_spec (prg E) C (mn, pTs) = {((md,rT),pTs')} |] ==>
         E⊢{C}a..mn({pTs'}ps)::rT"
```

— well-typed expression lists

```
  — cf. 15.11.???
| Nil: "E⊢[][::][]"
```

— cf. 15.11.???
```
| Cons:"[| E⊢e::T;
           E⊢es[::]Ts |] ==>
         E⊢e#es[::]T#Ts"
```

— well-typed statements

```
| Skip:"E⊢Skip√"

| Expr:"[| E⊢e::T |] ==>
         E⊢Expr e√"

| Comp:"[| E⊢s1√;
           E⊢s2√ |] ==>
         E⊢s1;; s2√"
```

— cf. 14.8
```
| Cond:"[| E⊢e::PrimT Boolean;
           E⊢s1√;
           E⊢s2√ |] ==>
          E⊢If(e) s1 Else s2√"
```

— cf. 14.10
```
| Loop:"[| E⊢e::PrimT Boolean;
           E⊢s√ |] ==>
         E⊢While(e) s√"
```

definition *wf_java_mdecl :: "'c prog => cname => java_mb mdecl => bool"* **where**
```
"wf_java_mdecl G C == λ((mn,pTs),rT,(pns,lvars,blk,res)).
  length pTs = length pns ∧
  distinct pns ∧
  unique lvars ∧
        This ∉ set pns ∧ This ∉ set (map fst lvars) ∧
  (∀pn∈set pns. map_of lvars pn = None) ∧
  (∀(vn,T)∈set lvars. is_type G T) &
  (let E = (G,(map_of lvars)(pns[↦]pTs, This↦Class C)) in
   E⊢blk√ ∧ (∃T. E⊢res::T ∧ G⊢T⪯rT))"
```

abbreviation *"wf_java_prog == wf_prog wf_java_mdecl"*

lemma *wf_java_prog_wf_java_mdecl: "⟦*
```
  wf_java_prog G; (C, D, fds, mths) ∈ set G; jmdcl ∈ set mths ⟧
  ⟹ wf_java_mdecl G C jmdcl"
```
⟨*proof*⟩

lemma *wt_is_type: "(E⊢e::T ⟶ ws_prog (prg E) ⟶ is_type (prg E) T) ∧*
```
       (E⊢es[::]Ts ⟶ ws_prog (prg E) ⟶ Ball (set Ts) (is_type (prg E))) ∧
       (E⊢c √ ⟶ True)"
```
⟨*proof*⟩

lemmas *ty_expr_is_type = wt_is_type [THEN conjunct1,THEN mp, rule_format]*

lemma *expr_class_is_class: "*
⟦*ws_prog (prg E); E ⊢ e :: Class C*⟧ ⟹ *is_class (prg E) C"*
⟨*proof*⟩

end

2.11 Operational Evaluation (big step) Semantics

theory *Eval* **imports** *State WellType* **begin**

— Auxiliary notions

definition *fits :: "java_mb prog ⇒ state ⇒ val ⇒ ty ⇒ bool" (<_,_⊢_ fits _>[61,61,61,61]60)*
where
"G,s⊢a' fits T ≡ case T of PrimT T' ⇒ False | RefT T' ⇒ a'=Null ∨ G⊢obj_ty(lookup_obj s a')≼T"

definition *catch :: "java_mb prog ⇒ xstate ⇒ cname ⇒ bool" (<_,_⊢catch _>[61,61,61]60)*
where
"G,s⊢catch C≡ case abrupt s of None ⇒ False | Some a ⇒ G,store s⊢ a fits Class C"

definition *lupd :: "vname ⇒ val ⇒ state ⇒ state" (<lupd'(_↦_')>[10,10]1000)* **where**
"lupd vn v ≡ λ (hp,loc). (hp, (loc(vn↦v)))"

definition *new_xcpt_var :: "vname ⇒ xstate ⇒ xstate"* **where**
"new_xcpt_var vn ≡ λ(x,s). Norm (lupd(vn↦the x) s)"

— Evaluation relations

inductive
eval :: "[java_mb prog,xstate,expr,val,xstate] => bool "
(<_ ⊢ _ -_≻_-> _> [51,82,60,82,82] 81)
and *evals :: "[java_mb prog,xstate,expr list,*
val list,xstate] => bool "
(<_ ⊢ _ -_[≻]_-> _> [51,82,60,51,82] 81)
and *exec :: "[java_mb prog,xstate,stmt, xstate] => bool "*
(<_ ⊢ _ -_-> _> [51,82,60,82] 81)
for *G :: "java_mb prog"*
where

— evaluation of expressions

XcptE:"G⊢(Some xc,s) -e≻undefined-> (Some xc,s)" — cf. 15.5

— cf. 15.8.1
| NewC: "[| h = heap s; (a,x) = new_Addr h;
h'= h(a↦(C,init_vars (fields (G,C)))) |] ==>
G⊢Norm s -NewC C≻Addr a-> c_hupd h' (x,s)"

```
  — cf. 15.15
| Cast: "[| G⊢Norm s0 -e≻v-> (x1,s1);
            x2 = raise_if (¬ cast_ok G C (heap s1) v) ClassCast x1 |] ==>
         G⊢Norm s0 -Cast C e≻v-> (x2,s1)"

  — cf. 15.7.1
| Lit:  "G⊢Norm s -Lit v≻v-> Norm s"

| BinOp:"[| G⊢Norm s -e1≻v1-> s1;
            G⊢s1     -e2≻v2-> s2;
            v = (case bop of Eq  => Bool (v1 = v2)
                           | Add => Intg (the_Intg v1 + the_Intg v2)) |] ==>
         G⊢Norm s -BinOp bop e1 e2≻v-> s2"

  — cf. 15.13.1, 15.2
| LAcc: "G⊢Norm s -LAcc v≻the (locals s v)-> Norm s"

  — cf. 15.25.1
| LAss: "[| G⊢Norm s -e≻v-> (x,(h,l));
            l' = (if x = None then l(va↦v) else l) |] ==>
         G⊢Norm s -va::=e≻v-> (x,(h,l'))"

  — cf. 15.10.1, 15.2
| FAcc: "[| G⊢Norm s0 -e≻a'-> (x1,s1);
            v = the (snd (the (heap s1 (the_Addr a')))(fn,T)) |] ==>
         G⊢Norm s0 -{T}e..fn≻v-> (np a' x1,s1)"

  — cf. 15.25.1
| FAss: "[| G⊢     Norm s0  -e1≻a'-> (x1,s1); a = the_Addr a';
            G⊢(np a' x1,s1) -e2≻v -> (x2,s2);
            h  = heap s2; (c,fs) = the (h a);
            h' = h(a↦(c,(fs((fn,T)↦v)))) |] ==>
         G⊢Norm s0 -{T}e1..fn:=e2≻v-> c_hupd h' (x2,s2)"

  — cf. 15.11.4.1, 15.11.4.2, 15.11.4.4, 15.11.4.5, 14.15
| Call: "[| G⊢Norm s0 -e≻a'-> s1; a = the_Addr a';
            G⊢s1 -ps[≻]pvs-> (x,(h,l)); dynT = fst (the (h a));
            (md,rT,pns,lvars,blk,res) = the (method (G,dynT) (mn,pTs));
            G⊢(np a' x,(h,(init_vars lvars)(pns[↦]pvs, This↦a'))) -blk-> s3;
            G⊢ s3 -res≻v -> (x4,s4) |] ==>
         G⊢Norm s0 -{C}e..mn({pTs}ps)≻v-> (x4,(heap s4,l))"


  — evaluation of expression lists

  — cf. 15.5
| XcptEs:"G⊢(Some xc,s) -e[≻]undefined-> (Some xc,s)"

  — cf. 15.11.???
| Nil:  "G⊢Norm s0 -[][≻][]-> Norm s0"

  — cf. 15.6.4
| Cons: "[| G⊢Norm s0 -e  ≻ v -> s1;
            G⊢     s1 -es[≻]vs-> s2 |] ==>
```

```
        G⊢Norm s0 -e#es[≻]v#vs-> s2"
```

— execution of statements

— cf. 14.1
```
| XcptS:"G⊢(Some xc,s) -c-> (Some xc,s)"
```

— cf. 14.5
```
| Skip: "G⊢Norm s -Skip-> Norm s"
```

— cf. 14.7
```
| Expr: "[| G⊢Norm s0 -e≻v-> s1 |] ==>
        G⊢Norm s0 -Expr e-> s1"
```

— cf. 14.2
```
| Comp: "[| G⊢Norm s0 -c1-> s1;
           G⊢     s1 -c2-> s2|] ==>
        G⊢Norm s0 -c1;; c2-> s2"
```

— cf. 14.8.2
```
| Cond: "[| G⊢Norm s0  -e≻v-> s1;
           G⊢ s1 -(if the_Bool v then c1 else c2)-> s2|] ==>
        G⊢Norm s0 -If(e) c1 Else c2-> s2"
```

— cf. 14.10, 14.10.1
```
| LoopF:"[| G⊢Norm s0 -e≻v-> s1; ¬the_Bool v |] ==>
        G⊢Norm s0 -While(e) c-> s1"
| LoopT:"[| G⊢Norm s0 -e≻v-> s1;  the_Bool v;
     G⊢s1 -c-> s2; G⊢s2 -While(e) c-> s3 |] ==>
        G⊢Norm s0 -While(e) c-> s3"
```

lemmas *eval_evals_exec_induct = eval_evals_exec.induct [split_format (complete)]*

lemma *NewCI: "[|new_Addr (heap s) = (a,x);*
s' = c_hupd ((heap s)(a↦(C,init_vars (fields (G,C))))) (x,s)|] ==>
G⊢Norm s -NewC C≻Addr a-> s'"
⟨*proof*⟩

lemma *eval_evals_exec_no_xcpt:*
"!!s s'. (G⊢(x,s) -e ≻ v -> (x',s') --> x'=None --> x=None) ∧
(G⊢(x,s) -es[≻]vs-> (x',s') --> x'=None --> x=None) ∧
(G⊢(x,s) -c   -> (x',s') --> x'=None --> x=None)"
⟨*proof*⟩

lemma *eval_no_xcpt: "G⊢(x,s) -e≻v-> (None,s') ==> x=None"*
⟨*proof*⟩

lemma *evals_no_xcpt: "G⊢(x,s) -e[≻]v-> (None,s') ==> x=None"*
⟨*proof*⟩

lemma *exec_no_xcpt: "G ⊢ (x, s) -c-> (None, s')*
⟹ x = None"

⟨*proof*⟩

lemma *eval_evals_exec_xcpt:*
```
"!!s s'. (G⊢(x,s) -e ≻  v -> (x',s') --> x=Some xc --> x'=Some xc ∧ s'=s) ∧
         (G⊢(x,s) -es[≻]vs-> (x',s') --> x=Some xc --> x'=Some xc ∧ s'=s) ∧
         (G⊢(x,s) -c       -> (x',s') --> x=Some xc --> x'=Some xc ∧ s'=s)"
```
⟨*proof*⟩

lemma *eval_xcpt:* `"G⊢(Some xc,s) -e≻v-> (x',s') ==> x'=Some xc ∧  s'=s"`
⟨*proof*⟩

lemma *exec_xcpt:* `"G⊢(Some xc,s) -s0-> (x',s') ==> x'=Some xc ∧  s'=s"`
⟨*proof*⟩

lemma *eval_LAcc_code:* `"G⊢Norm (h, l) -LAcc v≻the (l v)-> Norm (h, l)"`
⟨*proof*⟩

lemma *eval_Call_code:*
```
  "[| G⊢Norm s0 -e≻a'-> s1; a = the_Addr a';
      G⊢s1 -ps[≻]pvs-> (x,(h,l)); dynT = fst (the (h a));
      (md,rT,pns,lvars,blk,res) = the (method (G,dynT) (mn,pTs));
      G⊢(np a' x,(h,(init_vars lvars)(pns[↦]pvs, This↦a'))) -blk-> s3;
      G⊢ s3 -res≻v -> (x4,(h4, l4)) |] ==>
   G⊢Norm s0 -{C}e..mn({pTs}ps)≻v-> (x4,(h4,l))"
```
⟨*proof*⟩

lemmas *[code_pred_intro] = XcptE NewC Cast Lit BinOp*
lemmas *[code_pred_intro LAcc_code] = eval_LAcc_code*
lemmas *[code_pred_intro] = LAss FAcc FAss*
lemmas *[code_pred_intro Call_code] = eval_Call_code*
lemmas *[code_pred_intro] = XcptEs Nil Cons XcptS Skip Expr Comp Cond LoopF*
lemmas *[code_pred_intro LoopT_code] = LoopT*

code_pred
```
  (modes:
    eval: i ⇒ i ⇒ i ⇒ o ⇒ o ⇒ bool
  and
    evals: i ⇒ i ⇒ i ⇒ o ⇒ o ⇒ bool
  and
    exec: i ⇒ i ⇒ i ⇒ o ⇒ bool)
  eval
```
⟨*proof*⟩

end

theory *Exceptions* **imports** *State* **begin**

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

definition *blank :: "'c prog ⇒ cname ⇒ obj"* **where**
```
  "blank G C ≡ (C,init_vars (fields(G,C)))"
```

definition *start_heap :: "'c prog ⇒ aheap"* **where**
"start_heap G ≡ Map.empty (XcptRef NullPointer ↦ blank G (Xcpt NullPointer),
XcptRef ClassCast ↦ blank G (Xcpt ClassCast),
XcptRef OutOfMemory ↦ blank G (Xcpt OutOfMemory))"

abbreviation
cname_of :: "aheap ⇒ val ⇒ cname"
where *"cname_of hp v == fst (the (hp (the_Addr v)))"*

definition *preallocated :: "aheap ⇒ bool"* **where**
"preallocated hp ≡ ∀x. ∃fs. hp (XcptRef x) = Some (Xcpt x, fs)"

lemma *preallocatedD:*
"preallocated hp ⟹ ∃fs. hp (XcptRef x) = Some (Xcpt x, fs)"
⟨*proof*⟩

lemma *preallocatedE [elim?]:*
"preallocated hp ⟹ (⋀fs. hp (XcptRef x) = Some (Xcpt x, fs) ⟹ P hp) ⟹ P hp"
⟨*proof*⟩

lemma *cname_of_xcp:*
"raise_if b x None = Some xcp ⟹ preallocated hp
⟹ cname_of (hp::aheap) xcp = Xcpt x"
⟨*proof*⟩

lemma *preallocated_start:*
"preallocated (start_heap G)"
⟨*proof*⟩

end

2.12 Conformity Relations for Type Soundness Proof

theory *Conform* **imports** *State WellType Exceptions* **begin**

type_synonym *'c env' = "'c prog × (vname ⇀ ty)"* — same as *env* of *WellType.thy*

definition *hext :: "aheap => aheap => bool" (‹_ ≤| _› [51,51] 50)* **where**
"h≤|h' == ∀a C fs. h a = Some(C,fs) --> (∃fs'. h' a = Some(C,fs'))"

definition *conf :: "'c prog => aheap => val => ty => bool"*
(‹_,_ ⊢ _ ::≼ _› [51,51,51,51] 50) **where**
"G,h⊢v::≼T == ∃T'. typeof (map_option obj_ty o h) v = Some T' ∧ G⊢T'≼T"

definition *lconf :: "'c prog => aheap => ('a ⇀ val) => ('a ⇀ ty) => bool"*
(‹_,_ ⊢ _ [::≼] _› [51,51,51,51] 50) **where**
"G,h⊢vs[::≼]Ts == ∀n T. Ts n = Some T --> (∃v. vs n = Some v ∧ G,h⊢v::≼T)"

definition *oconf :: "'c prog => aheap => obj => bool" (‹_,_ ⊢ _ √› [51,51,51] 50)* **where**

```
"G,h⊢obj √ == G,h⊢snd obj[::≼]map_of (fields (G,fst obj))"
```

definition *hconf :: "'c prog => aheap => bool" (‹_ ⊢h _ √› [51,51] 50)* **where**
"G⊢h h √   == ∀a obj. h a = Some obj --> G,h⊢obj √"

definition *xconf :: "aheap ⇒ val option ⇒ bool"* **where**
"xconf hp vo == preallocated hp ∧ (∀ v. (vo = Some v) ⟶ (∃ xc. v = (Addr (XcptRef xc))))"

definition *conforms :: "xstate => java_mb env' => bool" (‹_ ::≼ _› [51,51] 50)* **where**
"s::≼E == prg E⊢h heap (store s) √ ∧
prg E,heap (store s)⊢locals (store s)[::≼]localT E ∧
xconf (heap (store s)) (abrupt s)"

2.12.1 hext

lemma *hextI:*
" ∀a C fs . h a = Some (C,fs) -->
(∃fs'. h' a = Some (C,fs')) ==> h≤|h'"
⟨*proof*⟩

lemma *hext_objD: "[|h≤|h'; h a = Some (C,fs) |] ==> ∃fs'. h' a = Some (C,fs')"*
⟨*proof*⟩

lemma *hext_refl [simp]: "h≤|h"*
⟨*proof*⟩

lemma *hext_new [simp]: "h a = None ==> h≤|h(a↦x)"*
⟨*proof*⟩

lemma *hext_trans: "[|h≤|h'; h'≤|h''|] ==> h≤|h''"*
⟨*proof*⟩

lemma *hext_upd_obj: "h a = Some (C,fs) ==> h≤|h(a↦(C,fs'))"*
⟨*proof*⟩

2.12.2 conf

lemma *conf_Null [simp]: "G,h⊢Null::≼T = G⊢RefT NullT≼T"*
⟨*proof*⟩

lemma *conf_litval [rule_format (no_asm), simp]:*
"typeof (λv. None) v = Some T --> G,h⊢v::≼T"
⟨*proof*⟩

lemma *conf_AddrI: "[|h a = Some obj; G⊢obj_ty obj≼T|] ==> G,h⊢Addr a::≼T"*
⟨*proof*⟩

lemma *conf_obj_AddrI: "[|h a = Some (C,fs); G⊢C≼C D|] ==> G,h⊢Addr a::≼ Class D"*
⟨*proof*⟩

lemma *defval_conf [rule_format (no_asm)]:*
"is_type G T --> G,h⊢default_val T::≼T"
⟨*proof*⟩

lemma *conf_upd_obj:*
"h a = Some (C,fs) ==> (G,h(a↦(C,fs'))⊢x::≼T) = (G,h⊢x::≼T)"
⟨*proof*⟩

lemma *conf_widen [rule_format (no_asm)]:*
"wf_prog wf_mb G ==> G,h⊢x::≼T --> G⊢T≼T' --> G,h⊢x::≼T'"
⟨*proof*⟩

lemma *conf_hext [rule_format (no_asm)]: "h≤|h' ==> G,h⊢v::≼T --> G,h'⊢v::≼T"*
⟨*proof*⟩

lemma *new_locD: "[|h a = None; G,h⊢Addr t::≼T|] ==> t≠a"*
⟨*proof*⟩

lemma *conf_RefTD [rule_format]:*
"G,h⊢a'::≼RefT T ⟹ a' = Null ∨
(∃a obj T'. a' = Addr a ∧ h a = Some obj ∧ obj_ty obj = T' ∧ G⊢T'≼RefT T)"
⟨*proof*⟩

lemma *conf_NullTD: "G,h⊢a'::≼RefT NullT ==> a' = Null"*
⟨*proof*⟩

lemma *non_npD: "[|a' ≠ Null; G,h⊢a'::≼RefT t|] ==>*
∃a C fs. a' = Addr a ∧ h a = Some (C,fs) ∧ G⊢Class C≼RefT t"
⟨*proof*⟩

lemma *non_np_objD: "!!G. [|a' ≠ Null; G,h⊢a'::≼ Class C|] ==>*
(∃a C' fs. a' = Addr a ∧ h a = Some (C',fs) ∧ G⊢C'≼C C)"
⟨*proof*⟩

lemma *non_np_objD' [rule_format (no_asm)]:*
"a' ≠ Null ==> wf_prog wf_mb G ==> G,h⊢a'::≼RefT t -->
(∃a C fs. a' = Addr a ∧ h a = Some (C,fs) ∧ G⊢Class C≼RefT t)"
⟨*proof*⟩

lemma *conf_list_gext_widen [rule_format (no_asm)]:*
"wf_prog wf_mb G ==> ∀Ts Ts'. list_all2 (conf G h) vs Ts -->
list_all2 (λT T'. G⊢T≼T') Ts Ts' --> list_all2 (conf G h) vs Ts'"
⟨*proof*⟩

2.12.3 lconf

lemma *lconfD: "[| G,h⊢vs[::≼]Ts; Ts n = Some T |] ==> G,h⊢(the (vs n))::≼T"*
⟨*proof*⟩

lemma *lconf_hext [elim]: "[| G,h⊢l[::≼]L; h≤|h' |] ==> G,h'⊢l[::≼]L"*
⟨*proof*⟩

lemma *lconf_upd: "!!X. [| G,h⊢l[::≼]lT;*
G,h⊢v::≼T; lT va = Some T |] ==> G,h⊢l(va↦v)[::≼]lT"
⟨*proof*⟩

lemma *lconf_init_vars_lemma [rule_format (no_asm)]:*

```
  "∀x. P x --> R (dv x) x ==> (∀x. map_of fs f = Some x --> P x) -->
  (∀T. map_of fs f = Some T -->
  (∃v. map_of (map (λ(f,ft). (f, dv ft)) fs) f = Some v ∧  R v T))"
```
⟨*proof*⟩

lemma *lconf_init_vars [intro!]:*
```
"∀n. ∀T. map_of fs n = Some T --> is_type G T ==> G,h⊢init_vars fs[::≼]map_of fs"
```
⟨*proof*⟩

lemma *lconf_ext:* `"[|G,s⊢l[::≼]L; G,s⊢v::≼T|] ==> G,s⊢l(vn↦v)[::≼]L(vn↦T)"`
⟨*proof*⟩

lemma *lconf_ext_list [rule_format (no_asm)]:*
```
  "G,h⊢l[::≼]L ==> ∀vs Ts. distinct vns --> length Ts = length vns -->
  list_all2 (λv T. G,h⊢v::≼T) vs Ts --> G,h⊢l(vns[↦]vs)[::≼]L(vns[↦]Ts)"
```
⟨*proof*⟩

lemma *lconf_restr:* `"⟦lT vn = None; G, h ⊢ l [::≼] lT(vn↦T)⟧ ⟹ G, h ⊢ l [::≼] lT"`
⟨*proof*⟩

2.12.4 oconf

lemma *oconf_hext:* `"G,h⊢obj√ ==> h≤|h' ==> G,h'⊢obj√"`
⟨*proof*⟩

lemma *oconf_obj:*
```
"G,h⊢(C,fs)√ =
  (∀T f. map_of(fields (G,C)) f = Some T --> (∃v. fs f = Some v ∧  G,h⊢v::≼T))"
```
⟨*proof*⟩

lemmas *oconf_objD = oconf_obj [THEN iffD1, THEN spec, THEN spec, THEN mp]*

2.12.5 hconf

lemma *hconfD:* `"[|G⊢h h√; h a = Some obj|] ==> G,h⊢obj√"`
⟨*proof*⟩

lemma *hconfI:* `"∀a obj. h a=Some obj --> G,h⊢obj√ ==> G⊢h h√"`
⟨*proof*⟩

2.12.6 xconf

lemma *xconf_raise_if:* `"xconf h x ⟹ xconf h (raise_if b xcn x)"`
⟨*proof*⟩

2.12.7 conforms

lemma *conforms_heapD:* `"(x, (h, l))::≼(G, lT) ==> G⊢h h√"`
⟨*proof*⟩

lemma *conforms_localD:* `"(x, (h, l))::≼(G, lT) ==> G,h⊢l[::≼]lT"`
⟨*proof*⟩

lemma *conforms_xcptD:* `"(x, (h, l))::≼(G, lT) ==> xconf h x"`
⟨*proof*⟩

lemma *conformsI: "[|G⊢h h√; G,h⊢l[::≼]lT; xconf h x|] ==> (x, (h, l))::≼(G, lT)"*
⟨*proof*⟩

lemma *conforms_restr: "⟦lT vn = None; s ::≼ (G, lT(vn↦T)) ⟧ ⟹ s ::≼ (G, lT)"*
⟨*proof*⟩

lemma *conforms_xcpt_change: "⟦ (x, (h,l))::≼ (G, lT); xconf h x ⟶ xconf h x' ⟧ ⟹*
(x', (h,l))::≼ (G, lT)"
⟨*proof*⟩

lemma *preallocated_hext: "⟦ preallocated h; h≤|h'⟧ ⟹ preallocated h'"*
⟨*proof*⟩

lemma *xconf_hext: "⟦ xconf h vo; h≤|h'⟧ ⟹ xconf h' vo"*
⟨*proof*⟩

lemma *conforms_hext: "[|(x,(h,l))::≼(G,lT); h≤|h'; G⊢h h'√ |]*
==> (x,(h',l))::≼(G,lT)"
⟨*proof*⟩

lemma *conforms_upd_obj:*
"[|(x,(h,l))::≼(G, lT); G,h(a↦obj)⊢obj√; h≤|h(a↦obj)|]
==> (x,(h(a↦obj),l))::≼(G, lT)"
⟨*proof*⟩

lemma *conforms_upd_local:*
"[|(x,(h, l))::≼(G, lT); G,h⊢v::≼T; lT va = Some T|]
==> (x,(h, l(va↦v)))::≼(G, lT)"
⟨*proof*⟩

end

2.13 Type Safety Proof

theory *JTypeSafe* **imports** *Eval Conform* **begin**

declare *split_beta [simp]*

lemma *NewC_conforms:*
"[|h a = None; (x,(h, l))::≼(G, lT); wf_prog wf_mb G; is_class G C|] ==>
(x,(h(a↦(C,(init_vars (fields (G,C))))), l))::≼(G, lT)"
⟨*proof*⟩

lemma *Cast_conf:*
"[| wf_prog wf_mb G; G,h⊢v::≼CC; G⊢CC ≼? Class D; cast_ok G D h v|]
==> G,h⊢v::≼Class D"
⟨*proof*⟩

lemma *FAcc_type_sound:*

```
"[| wf_prog wf_mb G; field (G,C) fn = Some (fd, ft); (x,(h,l))::≼(G,lT);
  x' = None --> G,h⊢a'::≼ Class C; np a' x' = None |] ==>
  G,h⊢the (snd (the (h (the_Addr a'))) (fn, fd))::≼ft"
```
⟨*proof*⟩

lemma *FAss_type_sound:*
```
 "[| wf_prog wf_mb G; a = the_Addr a'; (c, fs) = the (h a);
    (G, lT)⊢v::T'; G⊢T'≼ft;
    (G, lT)⊢aa::Class C;
    field (G,C) fn = Some (fd, ft); h''≤|h';
    x' = None --> G,h'⊢a'::≼ Class C; h'≤|h;
    Norm (h, l)::≼(G, lT); G,h⊢x::≼T'; np a' x' = None|] ==>
  h''≤|h(a↦(c,(fs((fn,fd)↦x)))) ∧
  Norm(h(a↦(c,(fs((fn,fd)↦x)))), l)::≼(G, lT) ∧
  G,h(a↦(c,(fs((fn,fd)↦x))))⊢x::≼T'"
```
⟨*proof*⟩

lemma *Call_lemma2:*
```
"[| wf_prog wf_mb G; list_all2 (conf G h) pvs pTs;
   list_all2 (λT T'. G⊢T≼T') pTs pTs'; wf_mhead G (mn,pTs') rT;
  length pTs' = length pns; distinct pns;
  Ball (set lvars) (case_prod (λvn. is_type G))
  |] ==> G,h⊢(init_vars lvars)(pns[↦]pvs)[::≼](map_of lvars)(pns[↦]pTs')"
```
⟨*proof*⟩

lemma *Call_type_sound:*
```
 "[| wf_java_prog G; a' ≠ Null; Norm (h, l)::≼(G, lT); class G C = Some y;
     max_spec G C (mn,pTsa) = {((mda,rTa),pTs')}; xc≤|xh; xh≤|h;
     list_all2 (conf G h) pvs pTsa;
     (md, rT, pns, lvars, blk, res) =
               the (method (G,fst (the (h (the_Addr a')))) (mn, pTs'));
  ∀lT. (np a' None, h, (init_vars lvars)(pns[↦]pvs, This↦a'))::≼(G, lT) -->
  (G, lT)⊢blk√ -->  h≤|xi ∧  (xcptb, xi, xl)::≼(G, lT);
  ∀lT. (xcptb,xi, xl)::≼(G, lT) --> (∀T. (G, lT)⊢res::T -->
         xi≤|h' ∧ (x',h', xj)::≼(G, lT) ∧ (x' = None --> G,h'⊢v::≼T));
  G,xh⊢a'::≼ Class C
  |] ==>
  xc≤|h' ∧ (x',(h', l))::≼(G, lT) ∧  (x' = None --> G,h'⊢v::≼rTa)"
```
⟨*proof*⟩

declare *if_split [split del]*
declare *fun_upd_apply [simp del]*
declare *fun_upd_same [simp]*
declare *wf_prog_ws_prog [simp]*

⟨*ML*⟩

theorem *eval_evals_exec_type_sound:*
```
"wf_java_prog G ==>
  (G⊢(x,(h,l)) -e ≻v -> (x', (h',l')) -->
```

```
      (∀ lT.   (x,(h ,l )):≼(G,lT) --> (∀ T . (G,lT)⊢e  :: T -->
      h≤|h' ∧ (x',(h',l')):≼(G,lT) ∧ (x'=None --> G,h'⊢v  ::≼ T )))) ∧
  (G⊢(x,(h,l)) -es[≻]vs-> (x', (h',l')) -->
      (∀ lT.   (x,(h ,l )):≼(G,lT) --> (∀ Ts. (G,lT)⊢es[::]Ts -->
      h≤|h' ∧ (x',(h',l')):≼(G,lT) ∧ (x'=None --> list_all2 (λv T. G,h'⊢v::≼T) vs Ts))))
∧
  (G⊢(x,(h,l)) -c       -> (x', (h',l')) -->
      (∀ lT.   (x,(h ,l )):≼(G,lT) -->       (G,lT)⊢c  √ -->
      h≤|h' ∧ (x',(h',l')):≼(G,lT)))"
```
⟨*proof*⟩

lemma *eval_type_sound: "!!E s s'.*
```
  [| wf_java_prog G; G⊢(x,s) -e≻v -> (x',s'); (x,s)::≼E; E⊢e::T; G=prg E |]
  ==> (x',s')::≼E ∧ (x'=None --> G,heap s'⊢v::≼T) ∧ heap s ≤| heap s'"
```
⟨*proof*⟩

lemma *evals_type_sound: "!!E s s'.*
```
  [| wf_java_prog G; G⊢(x,s) -es[≻]vs -> (x',s'); (x,s)::≼E; E⊢es[::]Ts; G=prg E |]
  ==> (x',s')::≼E ∧ (x'=None --> (list_all2 (λv T. G,heap s'⊢v::≼T) vs Ts)) ∧ heap s
≤| heap s'"
```
⟨*proof*⟩

lemma *exec_type_sound: "!!E s s'.*
```
  ⟦ wf_java_prog G; G⊢(x,s) -s0-> (x',s'); (x,s)::≼E; E⊢s0√; G=prg E ⟧
  ⟹ (x',s')::≼E ∧ heap s ≤| heap s'"
```
⟨*proof*⟩

theorem *all_methods_understood:*
```
"[|G=prg E; wf_java_prog G; G⊢(x,s) -e≻a'-> Norm s'; a' ≠ Null;
          (x,s)::≼E; E⊢e::Class C; method (G,C) sig ≠ None|] ==>
  method (G,fst (the (heap s' (the_Addr a')))) sig ≠ None"
```
⟨*proof*⟩

declare *split_beta [simp del]*
declare *fun_upd_apply [simp]*
declare *wf_prog_ws_prog [simp del]*

end

2.14 Example MicroJava Program

theory *Example* **imports** *SystemClasses Eval* **begin**

The following example MicroJava program includes: class declarations with inheritance, hiding of fields, and overriding of methods (with refined result type), instance creation, local assignment, sequential composition, method call with dynamic binding, literal values, expression statement, local access, type cast, field assignment (in part), skip.

```
class Base {
  boolean vee;
  Base foo(Base x) {return x;}
```

```
}

class Ext extends Base {
  int vee;
  Ext foo(Base x) {((Ext)x).vee=1; return null;}
}

class Example {
  public static void main (String args[]) {
    Base e=new Ext();
    e.foo(null);
  }
}
```

datatype *cnam' = Base' | Ext'*
datatype *vnam' = vee' | x' | e'*

cnam' and *vnam'* are intended to be isomorphic to *cnam* and *vnam*

axiomatization *cnam' :: "cnam' => cname"*
where
  inj_cnam': *"(cnam' x = cnam' y) = (x = y)"* **and**
  surj_cnam': *"∃m. n = cnam' m"*

axiomatization *vnam' :: "vnam' => vnam"*
where
  inj_vnam': *"(vnam' x = vnam' y) = (x = y)"* **and**
  surj_vnam': *"∃m. n = vnam' m"*

declare *inj_cnam' [simp] inj_vnam' [simp]*

abbreviation *Base :: cname*
  where *"Base == cnam' Base'"*
abbreviation *Ext :: cname*
  where *"Ext == cnam' Ext'"*
abbreviation *vee :: vname*
  where *"vee == VName (vnam' vee')"*
abbreviation *x :: vname*
  where *"x == VName (vnam' x')"*
abbreviation *e :: vname*
  where *"e == VName (vnam' e')"*

axiomatization where
  Base_not_Object: *"Base ≠ Object"* **and**
  Ext_not_Object: *"Ext ≠ Object"* **and**
  Base_not_Xcpt: *"Base ≠ Xcpt z"* **and**
  Ext_not_Xcpt: *"Ext ≠ Xcpt z"* **and**
  e_not_This: *"e ≠ This"*

declare *Base_not_Object [simp] Ext_not_Object [simp]*
declare *Base_not_Xcpt [simp] Ext_not_Xcpt [simp]*
declare *e_not_This [simp]*
declare *Base_not_Object [symmetric, simp]*

```
declare Ext_not_Object  [symmetric, simp]
declare Base_not_Xcpt [symmetric, simp]
declare Ext_not_Xcpt  [symmetric, simp]

definition foo_Base :: java_mb
  where "foo_Base == ([x],[],Skip,LAcc x)"

definition foo_Ext :: java_mb
  where "foo_Ext == ([x],[],Expr( {Ext}Cast Ext
              (LAcc x)..vee:=Lit (Intg Numeral1)),
          Lit Null)"

consts foo ::  mname

definition BaseC :: "java_mb cdecl"
  where "BaseC == (Base, (Object,
          [(vee, PrimT Boolean)],
          [((foo,[Class Base]),Class Base,foo_Base)]))"

definition ExtC :: "java_mb cdecl"
  where "ExtC  == (Ext,  (Base  ,
          [(vee, PrimT Integer)],
          [((foo,[Class Base]),Class Ext,foo_Ext)]))"

definition test ::  stmt
  where "test == Expr(e::=NewC Ext);;
                  Expr({Base}LAcc e..foo({[Class Base]}[Lit Null]))"

consts
  a :: loc
  b :: loc

abbreviation
  NP  :: xcpt where
  "NP == NullPointer"

abbreviation
  tprg  ::"java_mb prog" where
  "tprg == [ObjectC, BaseC, ExtC, ClassCastC, NullPointerC, OutOfMemoryC]"

abbreviation
  obj1  :: obj where
  "obj1 == (Ext, Map.empty((vee, Base)↦Bool False, (vee, Ext )↦Intg 0))"

abbreviation "s0 == Norm    (Map.empty, Map.empty)"
abbreviation "s1 == Norm    (Map.empty(a↦obj1),Map.empty(e↦Addr a))"
abbreviation "s2 == Norm    (Map.empty(a↦obj1),Map.empty(x↦Null, This↦Addr a))"
abbreviation "s3 == (Some NP, Map.empty(a↦obj1),Map.empty(e↦Addr a))"

lemmas map_of_Cons = map_of.simps(2)

lemma map_of_Cons1 [simp]: "map_of ((aa,bb)#ps) aa = Some bb"
⟨proof⟩
lemma map_of_Cons2 [simp]: "aa≠k ==> map_of ((k,bb)#ps) aa = map_of ps aa"
```

⟨*proof*⟩
declare *map_of_Cons [simp del]* — sic!

lemma *class_tprg_Object [simp]: "class tprg Object = Some (undefined, [], [])"*
⟨*proof*⟩

lemma *class_tprg_NP [simp]: "class tprg (Xcpt NP) = Some (Object, [], [])"*
⟨*proof*⟩

lemma *class_tprg_OM [simp]: "class tprg (Xcpt OutOfMemory) = Some (Object, [], [])"*
⟨*proof*⟩

lemma *class_tprg_CC [simp]: "class tprg (Xcpt ClassCast) = Some (Object, [], [])"*
⟨*proof*⟩

lemma *class_tprg_Base [simp]:*
"class tprg Base = Some (Object,
[(vee, PrimT Boolean)],
[((foo, [Class Base]), Class Base, foo_Base)])"
⟨*proof*⟩

lemma *class_tprg_Ext [simp]:*
"class tprg Ext = Some (Base,
[(vee, PrimT Integer)],
[((foo, [Class Base]), Class Ext, foo_Ext)])"
⟨*proof*⟩

lemma *not_Object_subcls [elim!]: "(Object, C) ∈ (subcls1 tprg)$^+$ ==> R"*
⟨*proof*⟩

lemma *subcls_ObjectD [dest!]: "tprg⊢Object≼C C ==> C = Object"*
⟨*proof*⟩

lemma *not_Base_subcls_Ext [elim!]: "(Base, Ext) ∈ (subcls1 tprg)$^+$ ==> R"*
⟨*proof*⟩

lemma *class_tprgD:*
"class tprg C = Some z ==> C=Object ∨ C=Base ∨ C=Ext ∨ C=Xcpt NP ∨ C=Xcpt ClassCast
∨ C=Xcpt OutOfMemory"
⟨*proof*⟩

lemma *not_class_subcls_class [elim!]: "(C, C) ∈ (subcls1 tprg)$^+$ ==> R"*
⟨*proof*⟩

lemma *unique_classes: "unique tprg"*
⟨*proof*⟩

lemmas *subcls_direct = subcls1I [THEN r_into_rtrancl [***where** *r="subcls1 G"]]* **for** *G*

lemma *Ext_subcls_Base [simp]: "tprg⊢Ext≼C Base"*
⟨*proof*⟩

lemma *Ext_widen_Base [simp]: "tprg⊢Class Ext≼ Class Base"*
⟨*proof*⟩

declare *ty_expr_ty_exprs_wt_stmt.intros [intro!]*

lemma *acyclic_subcls1': "acyclic (subcls1 tprg)"*
⟨*proof*⟩

lemmas *wf_subcls1' = acyclic_subcls1' [THEN finite_subcls1 [THEN finite_acyclic_wf_converse]]*

lemmas *fields_rec' = wf_subcls1' [THEN [2] fields_rec_lemma]*

lemma *fields_Object [simp]: "fields (tprg, Object) = []"*
⟨*proof*⟩

declare *is_class_def [simp]*

lemma *fields_Base [simp]: "fields (tprg,Base) = [((vee, Base), PrimT Boolean)]"*
⟨*proof*⟩

lemma *fields_Ext [simp]:*
"fields (tprg, Ext) = [((vee, Ext), PrimT Integer)] @ fields (tprg, Base)"
⟨*proof*⟩

lemmas *method_rec' = wf_subcls1' [THEN [2] method_rec_lemma]*

lemma *method_Object [simp]: "method (tprg,Object) = map_of []"*
⟨*proof*⟩

lemma *method_Base [simp]: "method (tprg, Base) = map_of*
[((foo, [Class Base]), Base, (Class Base, foo_Base))]"
⟨*proof*⟩

lemma *method_Ext [simp]: "method (tprg, Ext) = (method (tprg, Base) ++ map_of*
[((foo, [Class Base]), Ext , (Class Ext, foo_Ext))])"
⟨*proof*⟩

lemma *wf_foo_Base:*
"wf_mdecl wf_java_mdecl tprg Base ((foo, [Class Base]), (Class Base, foo_Base))"
⟨*proof*⟩

lemma *wf_foo_Ext:*
"wf_mdecl wf_java_mdecl tprg Ext ((foo, [Class Base]), (Class Ext, foo_Ext))"
⟨*proof*⟩

lemma *wf_ObjectC:*
"ws_cdecl tprg ObjectC ∧
wf_cdecl_mdecl wf_java_mdecl tprg ObjectC ∧ wf_mrT tprg ObjectC"
⟨*proof*⟩

lemma *wf_NP:*
"ws_cdecl tprg NullPointerC ∧
wf_cdecl_mdecl wf_java_mdecl tprg NullPointerC ∧ wf_mrT tprg NullPointerC"
⟨*proof*⟩

lemma *wf_OM:*

```
"ws_cdecl tprg OutOfMemoryC ∧
  wf_cdecl_mdecl wf_java_mdecl tprg OutOfMemoryC ∧ wf_mrT tprg OutOfMemoryC"
⟨proof⟩

lemma wf_CC:
"ws_cdecl tprg ClassCastC ∧
  wf_cdecl_mdecl wf_java_mdecl tprg ClassCastC ∧ wf_mrT tprg ClassCastC"
⟨proof⟩

lemma wf_BaseC:
"ws_cdecl tprg BaseC ∧
  wf_cdecl_mdecl wf_java_mdecl tprg BaseC ∧ wf_mrT tprg BaseC"
⟨proof⟩


lemma wf_ExtC:
"ws_cdecl tprg ExtC ∧
  wf_cdecl_mdecl wf_java_mdecl tprg ExtC ∧ wf_mrT tprg ExtC"
⟨proof⟩


lemma [simp]: "fst ObjectC = Object" ⟨proof⟩

lemma wf_tprg:
"wf_prog wf_java_mdecl tprg"
⟨proof⟩

lemma appl_methds_foo_Base:
"appl_methds tprg Base (foo, [NT]) =
  {((Class Base, Class Base), [Class Base])}"
⟨proof⟩

lemma max_spec_foo_Base: "max_spec tprg Base (foo, [NT]) =
  {((Class Base, Class Base), [Class Base])}"
⟨proof⟩

lemmas t = ty_expr_ty_exprs_wt_stmt.intros
schematic_goal wt_test: "(tprg, Map.empty(e↦Class Base))⊢
  Expr(e::=NewC Ext);; Expr({Base}LAcc e..foo({?pTs'}[Lit Null]))√"
⟨proof⟩


lemmas e = NewCI eval_evals_exec.intros
declare if_split [split del]
declare init_vars_def [simp] c_hupd_def [simp] cast_ok_def [simp]
schematic_goal exec_test:
" [|new_Addr (heap (snd s0)) = (a, None)|] ==>
  tprg⊢s0 -test-> ?s"
⟨proof⟩

end
```

2.15 Example for generating executable code from Java semantics

theory *JListExample*
imports *Eval*
begin

declare *[[syntax_ambiguity_warning = false]]*

consts
  list_nam :: cnam
  append_name :: mname

axiomatization *val_nam next_nam l_nam l1_nam l2_nam l3_nam l4_nam :: vnam*
where *distinct_fields: "val_nam ≠ next_nam"*
  and *distinct_vars1: "l_nam ≠ l1_nam"*
  and *distinct_vars2: "l_nam ≠ l2_nam"*
  and *distinct_vars3: "l_nam ≠ l3_nam"*
  and *distinct_vars4: "l_nam ≠ l4_nam"*
  and *distinct_vars5: "l1_nam ≠ l2_nam"*
  and *distinct_vars6: "l1_nam ≠ l3_nam"*
  and *distinct_vars7: "l1_nam ≠ l4_nam"*
  and *distinct_vars8: "l2_nam ≠ l3_nam"*
  and *distinct_vars9: "l2_nam ≠ l4_nam"*
  and *distinct_vars10: "l3_nam ≠ l4_nam"*

lemmas *distinct_vars =*
  distinct_vars1
  distinct_vars2
  distinct_vars3
  distinct_vars4
  distinct_vars5
  distinct_vars6
  distinct_vars7
  distinct_vars8
  distinct_vars9
  distinct_vars10

definition *list_name :: cname* **where**
  "list_name = Cname list_nam"

definition *val_name :: vname* **where**
  "val_name == VName val_nam"

definition *next_name :: vname* **where**
  "next_name == VName next_nam"

definition *l_name :: vname* **where**
  "l_name == VName l_nam"

definition *l1_name :: vname* **where**
  "l1_name == VName l1_nam"

definition *l2_name :: vname* **where**

```
  "l2_name == VName l2_nam"

definition l3_name :: vname where
  "l3_name == VName l3_nam"

definition l4_name :: vname where
  "l4_name == VName l4_nam"

definition list_class :: "java_mb class" where
  "list_class ==
    (Object,
     [(val_name, PrimT Integer), (next_name, RefT (ClassT list_name))],
     [((append_name, [RefT (ClassT list_name)]), PrimT Void,
      ([l_name], [],
       If(BinOp Eq ({list_name}(LAcc This)..next_name) (Lit Null))
         Expr ({list_name}(LAcc This)..next_name:=LAcc l_name)
       Else
         Expr ({list_name}({list_name}(LAcc This)..next_name)..
           append_name({[RefT (ClassT list_name)]}[LAcc l_name])),
       Lit Unit))])"

definition example_prg :: "java_mb prog" where
  "example_prg == [ObjectC, (list_name, list_class)]"

code_datatype list_nam
lemma equal_cnam_code [code]:
  "HOL.equal list_nam list_nam ⟷ True"
  ⟨proof⟩

code_datatype append_name
lemma equal_mname_code [code]:
  "HOL.equal append_name append_name ⟷ True"
  ⟨proof⟩

code_datatype val_nam next_nam l_nam l1_nam l2_nam l3_nam l4_nam
lemma equal_vnam_code [code]:
  "HOL.equal val_nam val_nam ⟷ True"
  "HOL.equal next_nam next_nam ⟷ True"
  "HOL.equal l_nam l_nam ⟷ True"
  "HOL.equal l1_nam l1_nam ⟷ True"
  "HOL.equal l2_nam l2_nam ⟷ True"
  "HOL.equal l3_nam l3_nam ⟷ True"
  "HOL.equal l4_nam l4_nam ⟷ True"

  "HOL.equal val_nam next_nam ⟷ False"
  "HOL.equal next_nam val_nam ⟷ False"

  "HOL.equal l_nam l1_nam ⟷ False"
  "HOL.equal l_nam l2_nam ⟷ False"
  "HOL.equal l_nam l3_nam ⟷ False"
  "HOL.equal l_nam l4_nam ⟷ False"

  "HOL.equal l1_nam l_nam ⟷ False"
  "HOL.equal l1_nam l2_nam ⟷ False"
```

```
  "HOL.equal l1_nam l3_nam ⟷ False"
  "HOL.equal l1_nam l4_nam ⟷ False"

  "HOL.equal l2_nam l_nam ⟷ False"
  "HOL.equal l2_nam l1_nam ⟷ False"
  "HOL.equal l2_nam l3_nam ⟷ False"
  "HOL.equal l2_nam l4_nam ⟷ False"

  "HOL.equal l3_nam l_nam ⟷ False"
  "HOL.equal l3_nam l1_nam ⟷ False"
  "HOL.equal l3_nam l2_nam ⟷ False"
  "HOL.equal l3_nam l4_nam ⟷ False"

  "HOL.equal l4_nam l_nam ⟷ False"
  "HOL.equal l4_nam l1_nam ⟷ False"
  "HOL.equal l4_nam l2_nam ⟷ False"
  "HOL.equal l4_nam l3_nam ⟷ False"
  ⟨proof⟩

axiomatization where
  nat_to_loc'_inject: "nat_to_loc' l = nat_to_loc' l' ⟷ l = l'"

lemma equal_loc'_code [code]:
  "HOL.equal (nat_to_loc' l) (nat_to_loc' l') ⟷ l = l'"
  ⟨proof⟩

definition undefined_cname :: cname
  where [code_abbrev]: "undefined_cname = undefined"
code_datatype Object Xcpt Cname undefined_cname

definition undefined_val :: val
  where [code_abbrev]: "undefined_val = undefined"
code_datatype Unit Null Bool Intg Addr undefined_val

definition E where
  "E = Expr (l1_name::=NewC list_name);;
       Expr ({list_name}(LAcc l1_name)..val_name:=Lit (Intg 1));;
       Expr (l2_name::=NewC list_name);;
       Expr ({list_name}(LAcc l2_name)..val_name:=Lit (Intg 2));;
       Expr (l3_name::=NewC list_name);;
       Expr ({list_name}(LAcc l3_name)..val_name:=Lit (Intg 3));;
       Expr (l4_name::=NewC list_name);;
       Expr ({list_name}(LAcc l4_name)..val_name:=Lit (Intg 4));;
       Expr ({list_name}(LAcc l1_name)..
         append_name({[RefT (ClassT list_name)]}[LAcc l2_name]));;
       Expr ({list_name}(LAcc l1_name)..
         append_name({[RefT (ClassT list_name)]}[LAcc l3_name]));;
       Expr ({list_name}(LAcc l1_name)..
         append_name({[RefT (ClassT list_name)]}[LAcc l4_name]))"

definition test where
  "test = Predicate.Pred (λs. example_prg⊢Norm (Map.empty, Map.empty) -E-> s)"

lemma test_code [code]:
```

```
  "test = exec_i_i_i_o example_prg (Norm (Map.empty, Map.empty)) E"
```
⟨*proof*⟩

⟨*ML*⟩

end

Chapter 3

Java Virtual Machine

3.1 State of the JVM

theory *JVMState*
imports *"../J/Conform"*
begin

3.1.1 Frame Stack

type_synonym *opstack = "val list"*
type_synonym *locvars = "val list"*
type_synonym *p_count = nat*

type_synonym
frame = "opstack ×
locvars ×
cname ×
sig ×
p_count"

— operand stack
— local variables (including this pointer and method parameters)
— name of class where current method is defined
— method name + parameter types
— program counter within frame

3.1.2 Exceptions

definition *raise_system_xcpt :: "bool ⇒ xcpt ⇒ val option"* **where**
"raise_system_xcpt b x ≡ raise_if b x None"

3.1.3 Runtime State

type_synonym
jvm_state = "val option × aheap × frame list" — exception flag, heap, frames

3.1.4 Lemmas

lemma *new_Addr_OutOfMemory:*
"snd (new_Addr hp) = Some xcp ⟹ xcp = Addr (XcptRef OutOfMemory)"
⟨*proof*⟩

end

3.2 Instructions of the JVM

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

datatype
```
  instr = Load nat                    — load from local variable
        | Store nat                   — store into local variable
        | LitPush val                 — push a literal (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 cname mname "(ty list)"  — inv. instance meth of an object
        | Return                      — return from method
        | Pop                         — pop top element from opstack
        | Dup                         — duplicate top element of opstack
        | Dup_x1                      — duplicate top element and push 2 down
        | Dup_x2                      — duplicate top element and push 3 down
        | Swap                        — swap top and next to top element
        | IAdd                        — integer addition
        | Goto int                    — goto relative address
        | Ifcmpeq int                 — branch if int/ref comparison succeeds
        | Throw                       — throw top of stack as exception
```

type_synonym
```
  bytecode = "instr list"
```
type_synonym
```
  exception_entry = "p_count × p_count × p_count × cname"
```
— start-pc, end-pc, handler-pc, exception type

type_synonym
```
  exception_table = "exception_entry list"
```
type_synonym
```
  jvm_method = "nat × nat × bytecode × exception_table"
```
— max stacksize, size of register set, instruction sequence, handler table

type_synonym
```
  jvm_prog = "jvm_method prog"
```

end

3.3 JVM Instruction Semantics

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

primrec *exec_instr :: "[instr, jvm_prog, aheap, opstack, locvars, cname, sig, p_count, frame list] => jvm_state"*
where
```
 "exec_instr (Load idx) G hp stk vars Cl sig pc frs =
      (None, hp, ((vars ! idx) # stk, vars, Cl, sig, pc+1)#frs)" |
```

```
"exec_instr (Store idx) G hp stk vars Cl sig pc frs =
     (None, hp, (tl stk, vars[idx:=hd stk], Cl, sig, pc+1)#frs)" |

"exec_instr (LitPush v) G hp stk vars Cl sig pc frs =
     (None, hp, (v # stk, vars, Cl, sig, pc+1)#frs)" |

"exec_instr (New C) G hp stk vars Cl sig pc frs =
 (let (oref,xp') = new_Addr hp;
      fs   = init_vars (fields(G,C));
      hp'  = if xp'=None then hp(oref ↦ (C,fs)) else hp;
      pc'  = if xp'=None then pc+1 else pc
  in
     (xp', hp', (Addr oref#stk, vars, Cl, sig, pc')#frs))" |

"exec_instr (Getfield F C) G hp stk vars Cl sig pc frs =
 (let oref = hd stk;
      xp'  = raise_system_xcpt (oref=Null) NullPointer;
      (oc,fs)  = the(hp(the_Addr oref));
      pc'  = if xp'=None then pc+1 else pc
  in
     (xp', hp, (the(fs(F,C))#(tl stk), vars, Cl, sig, pc')#frs))" |

"exec_instr (Putfield F C) G hp stk vars Cl sig pc frs =
 (let (fval,oref)= (hd stk, hd(tl stk));
      xp'  = raise_system_xcpt (oref=Null) NullPointer;
      a    = the_Addr oref;
      (oc,fs)  = the(hp a);
      hp'  = if xp'=None then hp(a ↦ (oc, fs((F,C) ↦ fval))) else hp;
      pc'  = if xp'=None then pc+1 else pc
  in
     (xp', hp', (tl (tl stk), vars, Cl, sig, pc')#frs))" |

"exec_instr (Checkcast C) G hp stk vars Cl sig pc frs =
 (let oref = hd stk;
      xp'  = raise_system_xcpt (¬ cast_ok G C hp oref) ClassCast;
      stk' = if xp'=None then stk else tl stk;
      pc'  = if xp'=None then pc+1 else pc
  in
     (xp', hp, (stk', vars, Cl, sig, pc')#frs))" |

"exec_instr (Invoke C mn ps) G hp stk vars Cl sig pc frs =
 (let n    = length ps;
      argsoref = take (n+1) stk;
      oref = last argsoref;
      xp'  = raise_system_xcpt (oref=Null) NullPointer;
      dynT = fst(the(hp(the_Addr oref)));
      (dc,mh,mxs,mxl,c)= the (method (G,dynT) (mn,ps));
      frs' = if xp'=None then
                [([],rev argsoref@replicate mxl undefined,dc,(mn,ps),0)]
             else []
  in
     (xp', hp, frs'@(stk, vars, Cl, sig, pc)#frs))" |
```

— Because exception handling needs the pc of the Invoke instruction,

— Invoke doesn't change stk and pc yet (`Return` does that).

```
"exec_instr Return G hp stk0 vars Cl sig0 pc frs =
 (if frs=[] then
    (None, hp, [])
  else
    let val = hd stk0; (stk,loc,C,sig,pc) = hd frs;
        (mn,pt) = sig0; n = length pt
  in
    (None, hp, (val#(drop (n+1) stk),loc,C,sig,pc+1)#tl frs))"
— Return drops arguments from the caller's stack and increases
— the program counter in the caller  |

"exec_instr Pop G hp stk vars Cl sig pc frs =
     (None, hp, (tl stk, vars, Cl, sig, pc+1)#frs)" |

"exec_instr Dup G hp stk vars Cl sig pc frs =
     (None, hp, (hd stk # stk, vars, Cl, sig, pc+1)#frs)" |

"exec_instr Dup_x1 G hp stk vars Cl sig pc frs =
     (None, hp, (hd stk # hd (tl stk) # hd stk # (tl (tl stk)),
                 vars, Cl, sig, pc+1)#frs)" |

"exec_instr Dup_x2 G hp stk vars Cl sig pc frs =
     (None, hp,
      (hd stk # hd (tl stk) # (hd (tl (tl stk))) # hd stk # (tl (tl (tl stk))),
      vars, Cl, sig, pc+1)#frs)" |

"exec_instr Swap G hp stk vars Cl sig pc frs =
 (let (val1,val2) = (hd stk,hd (tl stk))
  in
     (None, hp, (val2#val1#(tl (tl stk)), vars, Cl, sig, pc+1)#frs))" |

"exec_instr IAdd G hp stk vars Cl sig pc frs =
 (let (val1,val2) = (hd stk,hd (tl stk))
  in
     (None, hp, (Intg ((the_Intg val1)+(the_Intg val2))#(tl (tl stk)),
      vars, Cl, sig, pc+1)#frs))" |

"exec_instr (Ifcmpeq i) G hp stk vars Cl sig pc frs =
 (let (val1,val2) = (hd stk, hd (tl stk));
    pc' = if val1 = val2 then nat(int pc+i) else pc+1
  in
     (None, hp, (tl (tl stk), vars, Cl, sig, pc')#frs))" |

"exec_instr (Goto i) G hp stk vars Cl sig pc frs =
     (None, hp, (stk, vars, Cl, sig, nat(int pc+i))#frs)" |

"exec_instr Throw G hp stk vars Cl sig pc frs =
 (let xcpt  = raise_system_xcpt (hd stk = Null) NullPointer;
      xcpt' = if xcpt = None then Some (hd stk) else xcpt
  in
     (xcpt', hp, (stk, vars, Cl, sig, pc)#frs))"
```

end

3.4 Exception handling in the JVM

theory *JVMExceptions* **imports** *JVMInstructions* **begin**

```
definition match_exception_entry :: "jvm_prog ⇒ cname ⇒ p_count ⇒ exception_entry ⇒
bool" where
  "match_exception_entry G cn pc ee ==
                let (start_pc, end_pc, handler_pc, catch_type) = ee in
                start_pc <= pc ∧ pc < end_pc ∧ G⊢ cn ⪯C catch_type"


primrec match_exception_table :: "jvm_prog ⇒ cname ⇒ p_count ⇒ exception_table
    ⇒ p_count option"
where
  "match_exception_table G cn pc []     = None"
| "match_exception_table G cn pc (e#es) = (if match_exception_entry G cn pc e
                                            then Some (fst (snd (snd e)))
                                            else match_exception_table G cn pc es)"

abbreviation
  ex_table_of :: "jvm_method ⇒ exception_table"
  where "ex_table_of m == snd (snd (snd m))"


primrec find_handler :: "jvm_prog ⇒ val option ⇒ aheap ⇒ frame list
    ⇒ jvm_state"
where
  "find_handler G xcpt hp [] = (xcpt, hp, [])"
| "find_handler G xcpt hp (fr#frs) =
      (case xcpt of
         None ⇒ (None, hp, fr#frs)
       | Some xc ⇒
       let (stk,loc,C,sig,pc) = fr in
       (case match_exception_table G (cname_of hp xc) pc
              (ex_table_of (snd(snd(the(method (G,C) sig))))) of
          None ⇒ find_handler G (Some xc) hp frs
        | Some handler_pc ⇒ (None, hp, ([xc], loc, C, sig, handler_pc)#frs)))"
```

System exceptions are allocated in all heaps:

Only program counters that are mentioned in the exception table can be returned by `match_exception_table`:

lemma *match_exception_table_in_et:*
```
  "match_exception_table G C pc et = Some pc' ⟹ ∃e ∈ set et. pc' = fst (snd e))"
```
⟨*proof*⟩

end

3.5 Program Execution in the JVM

theory *JVMExec* **imports** *JVMExecInstr JVMExceptions* **begin**

fun
```
  exec :: "jvm_prog × jvm_state => jvm_state option"
```
— exec is not recursive. fun is just used for pattern matching
where
```
  "exec (G, xp, hp, []) = None"

| "exec (G, None, hp, (stk,loc,C,sig,pc)#frs) =
  (let
     i = fst(snd(snd(snd(snd(the(method (G,C) sig))))) ! pc;
     (xcpt', hp', frs') = exec_instr i G hp stk loc C sig pc frs
   in Some (find_handler G xcpt' hp' frs'))"

| "exec (G, Some xp, hp, frs) = None"
```

definition `exec_all :: "[jvm_prog,jvm_state,jvm_state] => bool"`
`(<_ ⊢ _ −jvm→ _> [61,61,61]60)` **where**
```
  "G ⊢ s −jvm→ t == (s,t) ∈ {(s,t). exec(G,s) = Some t}*"
```

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

definition `start_state :: "jvm_prog ⇒ cname ⇒ mname ⇒ jvm_state"` **where**
```
  "start_state G C m ≡
  let (C',rT,mxs,mxl,i,et) = the (method (G,C) (m,[])) in
    (None, start_heap G, [([], Null # replicate mxl undefined, C, (m,[]), 0)])"
```

end

3.6 Example for generating executable code from JVM semantics

theory *JVMListExample*
imports *"../J/SystemClasses" JVMExec*
begin

Since the types `cnam`, `vnam`, and `mname` are anonymous, we describe distinctness of names in the example by axioms:

axiomatization `list_nam test_nam :: cnam`
where `distinct_classes: "list_nam ≠ test_nam"`

axiomatization `append_name makelist_name :: mname`
where `distinct_methods: "append_name ≠ makelist_name"`

axiomatization `val_nam next_nam :: vnam`
where `distinct_fields: "val_nam ≠ next_nam"`

axiomatization
where `nat_to_loc'_inject: "nat_to_loc' l = nat_to_loc' l' ⟷ l = l'"`

definition `list_name :: cname`

```
  where "list_name = Cname list_nam"

definition test_name :: cname
  where "test_name = Cname test_nam"

definition val_name :: vname
  where "val_name = VName val_nam"

definition next_name :: vname
  where "next_name = VName next_nam"

definition append_ins :: bytecode where
  "append_ins =
       [Load 0,
        Getfield next_name list_name,
        Dup,
        LitPush Null,
        Ifcmpeq 4,
        Load 1,
        Invoke list_name append_name [Class list_name],
        Return,
        Pop,
        Load 0,
        Load 1,
        Putfield next_name list_name,
        LitPush Unit,
        Return]"

definition list_class :: "jvm_method class" where
  "list_class =
    (Object,
     [(val_name, PrimT Integer), (next_name, Class list_name)],
     [((append_name, [Class list_name]), PrimT Void,
        (3, 0, append_ins,[(1,2,8,Xcpt NullPointer)]))])"

definition make_list_ins :: bytecode where
  "make_list_ins =
       [New list_name,
        Dup,
        Store 0,
        LitPush (Intg 1),
        Putfield val_name list_name,
        New list_name,
        Dup,
        Store 1,
        LitPush (Intg 2),
        Putfield val_name list_name,
        New list_name,
        Dup,
        Store 2,
        LitPush (Intg 3),
        Putfield val_name list_name,
        Load 0,
        Load 1,
```

```
        Invoke list_name append_name [Class list_name],
        Pop,
        Load 0,
        Load 2,
        Invoke list_name append_name [Class list_name],
        Return]"

definition test_class :: "jvm_method class" where
  "test_class =
    (Object, [],
     [((makelist_name, []), PrimT Void, (3, 2, make_list_ins,[]))])"

definition E :: jvm_prog where
  "E = SystemClasses @ [(list_name, list_class), (test_name, test_class)]"

code_datatype list_nam test_nam
lemma equal_cnam_code [code]:
  "HOL.equal list_nam list_nam ⟷ True"
  "HOL.equal test_nam test_nam ⟷ True"
  "HOL.equal list_nam test_nam ⟷ False"
  "HOL.equal test_nam list_nam ⟷ False"
  ⟨proof⟩

code_datatype append_name makelist_name
lemma equal_mname_code [code]:
  "HOL.equal append_name append_name ⟷ True"
  "HOL.equal makelist_name makelist_name ⟷ True"
  "HOL.equal append_name makelist_name ⟷ False"
  "HOL.equal makelist_name append_name ⟷ False"
  ⟨proof⟩

code_datatype val_nam next_nam
lemma equal_vnam_code [code]:
  "HOL.equal val_nam val_nam ⟷ True"
  "HOL.equal next_nam next_nam ⟷ True"
  "HOL.equal val_nam next_nam ⟷ False"
  "HOL.equal next_nam val_nam ⟷ False"
  ⟨proof⟩


lemma equal_loc'_code [code]:
  "HOL.equal (nat_to_loc' l) (nat_to_loc' l') ⟷ l = l'"
  ⟨proof⟩

definition undefined_cname :: cname
  where [code_abbrev]: "undefined_cname = undefined"
code_datatype Object Xcpt Cname undefined_cname

definition undefined_val :: val
  where [code_abbrev]: "undefined_val = undefined"
code_datatype Unit Null Bool Intg Addr undefined_val

definition
  "test = exec (E, start_state E test_name makelist_name)"
```

⟨*ML*⟩

end

3.7 A Defensive JVM

theory *JVMDefensive*
imports *JVMExec*
begin

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

datatype *'a type_error = TypeError | Normal 'a*

abbreviation
fifth :: "'a × 'b × 'c × 'd × 'e × 'f ⇒ 'e"
where *"fifth x == fst(snd(snd(snd(snd x))))"*

fun *isAddr :: "val ⇒ bool"* **where**
"isAddr (Addr loc) = True"
| "isAddr v = False"

fun *isIntg :: "val ⇒ bool"* **where**
"isIntg (Intg i) = True"
| "isIntg v = False"

definition *isRef :: "val ⇒ bool"* **where**
"isRef v ≡ v = Null ∨ isAddr v"

primrec *check_instr :: "[instr, jvm_prog, aheap, opstack, locvars,*
cname, sig, p_count, nat, frame list] ⇒ bool" **where**
"check_instr (Load idx) G hp stk vars C sig pc mxs frs =
(idx < length vars ∧ size stk < mxs)"

| "check_instr (Store idx) G hp stk vars Cl sig pc mxs frs =
(0 < length stk ∧ idx < length vars)"

| "check_instr (LitPush v) G hp stk vars Cl sig pc mxs frs =
(¬isAddr v ∧ size stk < mxs)"

| "check_instr (New C) G hp stk vars Cl sig pc mxs frs =
(is_class G C ∧ size stk < mxs)"

| "check_instr (Getfield F C) G hp stk vars Cl sig pc mxs frs =
(0 < length stk ∧ is_class G C ∧ field (G,C) F ≠ None ∧
(let (C', T) = the (field (G,C) F); ref = hd stk in
C' = C ∧ isRef ref ∧ (ref ≠ Null ⟶
hp (the_Addr ref) ≠ None ∧
(let (D,vs) = the (hp (the_Addr ref)) in
G ⊢ D ≼C C ∧ vs (F,C) ≠ None ∧ G,hp ⊢ the (vs (F,C)) ::≼ T))))"

| "check_instr (Putfield F C) G hp stk vars Cl sig pc mxs frs =

```
  (1 < length stk ∧ is_class G C ∧ field (G,C) F ≠ None ∧
  (let (C', T) = the (field (G,C) F); v = hd stk; ref = hd (tl stk) in
    C' = C ∧ isRef ref ∧ (ref ≠ Null ⟶
      hp (the_Addr ref) ≠ None ∧
      (let (D,vs) = the (hp (the_Addr ref)) in
        G ⊢ D ⪯C C ∧ G,hp ⊢ v ::⪯ T))))"

| "check_instr (Checkcast C) G hp stk vars Cl sig pc mxs frs =
  (0 < length stk ∧ is_class G C ∧ isRef (hd stk))"

| "check_instr (Invoke C mn ps) G hp stk vars Cl sig pc mxs frs =
  (length ps < length stk ∧
  (let n = length ps; v = stk!n in
  isRef v ∧ (v ≠ Null ⟶
    hp (the_Addr v) ≠ None ∧
    method (G,cname_of hp v) (mn,ps) ≠ None ∧
    list_all2 (λv T. G,hp ⊢ v ::⪯ T) (rev (take n stk)) ps)))"

| "check_instr Return G hp stk0 vars Cl sig0 pc mxs frs =
  (0 < length stk0 ∧ (0 < length frs ⟶
    method (G,Cl) sig0 ≠ None ∧
    (let v = hd stk0;  (C, rT, body) = the (method (G,Cl) sig0) in
    Cl = C ∧ G,hp ⊢ v ::⪯ rT)))"

| "check_instr Pop G hp stk vars Cl sig pc mxs frs =
  (0 < length stk)"

| "check_instr Dup G hp stk vars Cl sig pc mxs frs =
  (0 < length stk ∧ size stk < mxs)"

| "check_instr Dup_x1 G hp stk vars Cl sig pc mxs frs =
  (1 < length stk ∧ size stk < mxs)"

| "check_instr Dup_x2 G hp stk vars Cl sig pc mxs frs =
  (2 < length stk ∧ size stk < mxs)"

| "check_instr Swap G hp stk vars Cl sig pc mxs frs =
  (1 < length stk)"

| "check_instr IAdd G hp stk vars Cl sig pc mxs frs =
  (1 < length stk ∧ isIntg (hd stk) ∧ isIntg (hd (tl stk)))"

| "check_instr (Ifcmpeq b) G hp stk vars Cl sig pc mxs frs =
  (1 < length stk ∧ 0 ≤ int pc+b)"

| "check_instr (Goto b) G hp stk vars Cl sig pc mxs frs =
  (0 ≤ int pc+b)"

| "check_instr Throw G hp stk vars Cl sig pc mxs frs =
  (0 < length stk ∧ isRef (hd stk))"

definition check :: "jvm_prog ⇒ jvm_state ⇒ bool" where
  "check G s ≡ let (xcpt, hp, frs) = s in
              (case frs of [] ⇒ True | (stk,loc,C,sig,pc)#frs' ⇒
```

```
                (let  (C',rt,mxs,mxl,ins,et) = the (method (G,C) sig); i = ins!pc in
                 pc < size ins ∧
                 check_instr i G hp stk loc C sig pc mxs frs'))"
```

definition *exec_d :: "jvm_prog ⇒ jvm_state type_error ⇒ jvm_state option type_error"*
where
```
  "exec_d G s ≡ case s of
      TypeError ⇒ TypeError
    | Normal s' ⇒ if check G s' then Normal (exec (G, s')) else TypeError"
```

definition
```
  exec_all_d :: "jvm_prog ⇒ jvm_state type_error ⇒ jvm_state type_error ⇒ bool"
                  (‹_ ⊢ _ −jvmd→ _› [61,61,61]60) where
  "G ⊢ s −jvmd→ t ⟷
         (s,t) ∈ ({(s,t). exec_d G s = TypeError ∧ t = TypeError} ∪
                  {(s,t). ∃t'. exec_d G s = Normal (Some t') ∧ t = Normal t'})*"
```

declare *split_paired_All [simp del]*
declare *split_paired_Ex [simp del]*

lemma *[dest!]:*
```
  "(if P then A else B) ≠ B ⟹ P"
```
⟨*proof*⟩

lemma *exec_d_no_errorI [intro]:*
```
  "check G s ⟹ exec_d G (Normal s) ≠ TypeError"
```
⟨*proof*⟩

theorem *no_type_error_commutes:*
```
  "exec_d G (Normal s) ≠ TypeError ⟹
  exec_d G (Normal s) = Normal (exec (G, s))"
```
⟨*proof*⟩

lemma *defensive_imp_aggressive:*
```
  "G ⊢ (Normal s) −jvmd→ (Normal t) ⟹ G ⊢ s −jvm→ t"
```
⟨*proof*⟩

end

Chapter 4

Bytecode Verifier

4.1 Semilattices

theory *Semilat*
imports *Main "HOL-Library.While_Combinator"*
begin

type_synonym *'a ord = "'a ⇒ 'a ⇒ bool"*
type_synonym *'a binop = "'a ⇒ 'a ⇒ 'a"*
type_synonym *'a sl = "'a set × 'a ord × 'a binop"*

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

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

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

notation *(ASCII)*
  "lesub" (‹(_ /<='__ _)› [50, 1000, 51] 50) **and**
  "lesssub" (‹(_ /<'__ _)› [50, 1000, 51] 50) **and**
  "plussub" (‹(_ /+'__ _)› [65, 1000, 66] 65)

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

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

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

abbreviation *(input)*

plussub1 :: "'a ⇒ ('a ⇒ 'b ⇒ 'c) ⇒ 'b ⇒ 'c" (‹(_ /⊔_ _)› [65, 1000, 66] 65)
where *"x ⊔$_f$ y == x ⊔$_f$ 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 ⊑$_r$ z ⟶ x ⊔$_f$ y ⊑$_r$ z)"

definition *is_ub :: "('a × 'a) set ⇒ 'a ⇒ 'a ⇒ 'a ⇒ bool"* **where**
"is_ub r x y u ≡ (x,u)∈r ∧ (y,u)∈r"

definition *is_lub :: "('a × 'a) set ⇒ 'a ⇒ 'a ⇒ 'a ⇒ bool"* **where**
"is_lub r x y u ≡ is_ub r x y u ∧ (∀z. is_ub r x y z ⟶ (u,z)∈r)"

definition *some_lub :: "('a × 'a) set ⇒ 'a ⇒ 'a ⇒ 'a"* **where**
"some_lub r x y ≡ SOME z. is_lub r x y z"

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

lemma *order_refl [simp, intro]: "order r ⟹ x ⊑$_r$ x"*
⟨*proof*⟩

lemma *order_antisym: "⟦ order r; x ⊑$_r$ y; y ⊑$_r$ x ⟧ ⟹ x = y"*
⟨*proof*⟩

lemma *order_trans: "⟦ order r; x ⊑$_r$ y; y ⊑$_r$ z ⟧ ⟹ x ⊑$_r$ z"*
⟨*proof*⟩

lemma *order_less_irrefl [intro, simp]: "order r ⟹ ¬ x ⊏$_r$ x"*
⟨*proof*⟩

lemma *order_less_trans: "⟦ order r; x ⊏$_r$ y; y ⊏$_r$ z ⟧ ⟹ x ⊏$_r$ z"*
⟨*proof*⟩

lemma *topD [simp, intro]: "top r T ⟹ x ⊑$_r$ T"*
⟨*proof*⟩

lemma *top_le_conv [simp]: "⟦ order r; top r T ⟧ ⟹ (T ⊑$_r$ x) = (x = T)"*
⟨*proof*⟩

lemma *semilat_Def:*
"semilat(A,r,f) ≡ order r ∧
(∀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 ⊑$_r$ z ⟶ x ⊔$_f$ y ⊑$_r$ z)"
⟨*proof*⟩

lemma (**in** *Semilat*) *orderI [simp, intro]: "order r"*
⟨*proof*⟩

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

lemma *closedD: "⟦ closed A f; x∈A; y∈A ⟧ ⟹ x ⊔$_f$ y ∈ A"*
⟨*proof*⟩

lemma *closed_UNIV [simp]: "closed UNIV f"*
⟨*proof*⟩

lemma (**in** *Semilat*) *closed_f [simp, intro]: "⟦x ∈ A; y ∈ A⟧ ⟹ x ⊔$_f$ y ∈ A"*
⟨*proof*⟩

lemma (**in** *Semilat*) *refl_r [intro, simp]: "x ⊑$_r$ x"* ⟨*proof*⟩

lemma (**in** *Semilat*) *antisym_r [intro?]: "⟦ x ⊑$_r$ y; y ⊑$_r$ x ⟧ ⟹ x = y"*
⟨*proof*⟩

lemma (**in** *Semilat*) *trans_r [trans, intro?]: "⟦x ⊑$_r$ y; y ⊑$_r$ z⟧ ⟹ x ⊑$_r$ z"*
⟨*proof*⟩

lemma (**in** *Semilat*) *ub1 [simp, intro?]: "⟦ x ∈ A; y ∈ A ⟧ ⟹ x ⊑$_r$ x ⊔$_f$ y"*
⟨*proof*⟩

lemma (**in** *Semilat*) *ub2 [simp, intro?]: "⟦ x ∈ A; y ∈ A ⟧ ⟹ y ⊑$_r$ x ⊔$_f$ y"*
⟨*proof*⟩

lemma (**in** *Semilat*) *lub [simp, intro?]:*
"⟦ x ⊑$_r$ z; y ⊑$_r$ z; x ∈ A; y ∈ A; z ∈ A ⟧ ⟹ x ⊔$_f$ y ⊑$_r$ z"
⟨*proof*⟩

lemma (**in** *Semilat*) *plus_le_conv [simp]:*
"⟦ x ∈ A; y ∈ A; z ∈ A ⟧ ⟹ (x ⊔$_f$ y ⊑$_r$ z) = (x ⊑$_r$ z ∧ y ⊑$_r$ z)"
⟨*proof*⟩

lemma (**in** *Semilat*) *le_iff_plus_unchanged: "⟦ x ∈ A; y ∈ A ⟧ ⟹ (x ⊑$_r$ y) = (x ⊔$_f$ y = y)"*⟨*proof*⟩

lemma (**in** *Semilat*) *le_iff_plus_unchanged2*: "⟦ *x* ∈ *A*; *y* ∈ *A* ⟧ ⟹ (*x* ⊑$_r$ *y*) = (*y* ⊔$_f$ *x* = *y*)"⟨*proof*⟩

lemma (**in** *Semilat*) *plus_assoc* [*simp*]:
assumes *a*: "*a* ∈ *A*" **and** *b*: "*b* ∈ *A*" **and** *c*: "*c* ∈ *A*"
shows "*a* ⊔$_f$ (*b* ⊔$_f$ *c*) = *a* ⊔$_f$ *b* ⊔$_f$ *c*"⟨*proof*⟩

lemma (**in** *Semilat*) *plus_com_lemma*:
"⟦*a* ∈ *A*; *b* ∈ *A*⟧ ⟹ *a* ⊔$_f$ *b* ⊑$_r$ *b* ⊔$_f$ *a*"⟨*proof*⟩

lemma (**in** *Semilat*) *plus_commutative*:
"⟦*a* ∈ *A*; *b* ∈ *A*⟧ ⟹ *a* ⊔$_f$ *b* = *b* ⊔$_f$ *a*"
⟨*proof*⟩

lemma *is_lubD*:
"*is_lub r x y u* ⟹ *is_ub r x y u* ∧ (∀*z*. *is_ub r x y z* ⟶ (*u*,*z*) ∈ *r*)"
⟨*proof*⟩

lemma *is_ubI*:
"⟦ (*x*,*u*) ∈ *r*; (*y*,*u*) ∈ *r* ⟧ ⟹ *is_ub r x y u*"
⟨*proof*⟩

lemma *is_ubD*:
"*is_ub r x y u* ⟹ (*x*,*u*) ∈ *r* ∧ (*y*,*u*) ∈ *r*"
⟨*proof*⟩

lemma *is_lub_bigger1* [*iff*]:
"*is_lub* (*r**) *x y y* = ((*x*,*y*)∈*r**)"⟨*proof*⟩

lemma *is_lub_bigger2* [*iff*]:
"*is_lub* (*r**) *x y x* = ((*y*,*x*)∈*r**)"⟨*proof*⟩

lemma *extend_lub*:
"⟦ *single_valued r*; *is_lub* (*r**) *x y u*; (*x'*,*x*) ∈ *r* ⟧
⟹ ∃*v*. *is_lub* (*r**) *x'* *y v*"⟨*proof*⟩

lemma *single_valued_has_lubs* [*rule_format*]:
"⟦*single_valued r*; (*x*,*u*) ∈ *r**⟧ ⟹ (∀*y*. (*y*,*u*) ∈ *r** ⟶
(∃*z*. *is_lub* (*r**) *x y z*))"⟨*proof*⟩

lemma *some_lub_conv*:
"⟦*acyclic r*; *is_lub* (*r**) *x y u*⟧ ⟹ *some_lub* (*r**) *x y* = *u*"⟨*proof*⟩

lemma *is_lub_some_lub*:
"⟦*single_valued r*; *acyclic r*; (*x*,*u*)∈*r**; (*y*,*u*)∈*r**⟧
⟹ *is_lub* (*r**) *x y* (*some_lub* (*r**) *x y*)"
⟨*proof*⟩

4.1.1 An executable lub-finder

definition *exec_lub* :: "('*a* * '*a*) *set* ⇒ ('*a* ⇒ '*a*) ⇒ '*a binop*" **where**
"*exec_lub r f x y* ≡ *while* (λ*z*. (*x*,*z*) ∉ *r**) *f y*"

lemma *exec_lub_refl: "exec_lub r f T T = T"*
⟨*proof*⟩

lemma *acyclic_single_valued_finite:*
"⟦acyclic r; single_valued r; (x,y) ∈ r⁺*⟧*
⟹ finite (r ∩ {a. (x, a) ∈ r⁺*} × {b. (b, y) ∈ r*⁺*})"*⟨*proof*⟩

lemma *exec_lub_conv:*
"⟦ acyclic r; ∀x y. (x,y) ∈ r ⟶ f x = y; is_lub (r⁺*) x y u ⟧ ⟹*
exec_lub r f x y = u"⟨*proof*⟩

lemma *is_lub_exec_lub:*
"⟦ single_valued r; acyclic r; (x,u)∈r⁺*; (y,u)∈r*⁺*; ∀x y. (x,y) ∈ r ⟶ f x = y ⟧*
⟹ is_lub (r⁺*) x y (exec_lub r f x y)"*
⟨*proof*⟩

end

4.2 The Error Type

theory *Err*
imports *Semilat*
begin

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

type_synonym *'a ebinop = "'a ⇒ 'a ⇒ 'a err"*
type_synonym *'a esl = "'a set * 'a ord * 'a ebinop"*

primrec *ok_val :: "'a err ⇒ 'a"* **where**
"ok_val (OK x) = x"

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

definition *lift2 :: "('a ⇒ 'b ⇒ 'c err) ⇒ 'a err ⇒ 'b err ⇒ 'c err"* **where**
"lift2 f e1 e2 ==
case e1 of Err ⇒ Err
| OK x ⇒ (case e2 of Err ⇒ Err | OK y ⇒ f x y)"

definition *le :: "'a ord ⇒ 'a err ord"* **where**
"le r e1 e2 ==
case e2 of Err ⇒ True |
OK y ⇒ (case e1 of Err ⇒ False | OK x ⇒ x <=_r y)"

definition *sup :: "('a ⇒ 'b ⇒ 'c) ⇒ ('a err ⇒ 'b err ⇒ 'c err)"* **where**
"sup f == lift2(%x y. OK(x +_f y))"

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

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

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

abbreviation
err_semilat :: "'a esl ⇒ bool"
where *"err_semilat L == semilat(Err.sl L)"*

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

lemma *strict_Some [simp]:*
"(strict f x = OK y) = (∃ z. x = OK z ∧ f z = OK y)"
⟨*proof*⟩

lemma *not_Err_eq:*
"(x ≠ Err) = (∃a. x = OK a)"
⟨*proof*⟩

lemma *not_OK_eq:*
"(∀y. x ≠ OK y) = (x = Err)"
⟨*proof*⟩

lemma *unfold_lesub_err:*
"e1 <=_(le r) e2 == le r e1 e2"
⟨*proof*⟩

lemma *le_err_refl:*
"∀x. x <=_r x ⟹ e <=_(Err.le r) e"
⟨*proof*⟩

lemma *le_err_trans [rule_format]:*
"order r ⟹ e1 <=_(le r) e2 ⟶ e2 <=_(le r) e3 ⟶ e1 <=_(le r) e3"
⟨*proof*⟩

lemma *le_err_antisym [rule_format]:*
"order r ⟹ e1 <=_(le r) e2 ⟶ e2 <=_(le r) e1 ⟶ e1=e2"
⟨*proof*⟩

lemma *OK_le_err_OK:*
"(OK x <=_(le r) OK y) = (x <=_r y)"
⟨*proof*⟩

lemma *order_le_err [iff]:*
"order(le r) = order r"
⟨*proof*⟩

lemma *le_Err [iff]: "e <=_(le r) Err"*
⟨*proof*⟩

lemma *Err_le_conv [iff]:*
"Err <=_(le r) e = (e = Err)"
⟨*proof*⟩

lemma *le_OK_conv [iff]:*
"e <=_(le r) OK x = (∃y. e = OK y & y <=_r x)"
⟨*proof*⟩

lemma *OK_le_conv:*
"OK x <=_(le r) e = (e = Err | (∃y. e = OK y & x <=_r y))"
⟨*proof*⟩

lemma *top_Err [iff]: "top (le r) Err"*
⟨*proof*⟩

lemma *OK_less_conv [rule_format, iff]:*
"OK x <_(le r) e = (e=Err | (∃y. e = OK y & x <_r y))"
⟨*proof*⟩

lemma *not_Err_less [rule_format, iff]:*
"~(Err <_(le r) x)"
⟨*proof*⟩

lemma *semilat_errI [intro]:*
assumes *semilat: "semilat (A, r, f)"*
shows *"semilat(err A, Err.le r, lift2(%x y. OK(f x y)))"*
⟨*proof*⟩

lemma *err_semilat_eslI_aux:*
assumes *semilat: "semilat (A, r, f)"*
shows *"err_semilat(esl(A,r,f))"*
⟨*proof*⟩

lemma *err_semilat_eslI [intro, simp]:*
"⋀L. semilat L ⟹ err_semilat(esl L)"
⟨*proof*⟩

lemma *acc_err [simp, intro!]: "acc r ⟹ acc(le r)"*
⟨*proof*⟩

lemma *Err_in_err [iff]: "Err ∈ err A"*
⟨*proof*⟩

lemma *Ok_in_err [iff]: "(OK x ∈ err A) = (x∈A)"*
⟨*proof*⟩

4.2.1 lift

lemma *lift_in_errI:*
"⟦ e ∈ err S; ∀x∈S. e = OK x ⟶ f x ∈ err S ⟧ ⟹ lift f e ∈ err S"
⟨*proof*⟩

lemma *Err_lift2 [simp]:*
"Err +_(lift2 f) x = Err"

⟨*proof*⟩

lemma *lift2_Err [simp]:*
"x +_(lift2 f) Err = Err"
⟨*proof*⟩

lemma *OK_lift2_OK [simp]:*
"OK x +_(lift2 f) OK y = x +_f y"
⟨*proof*⟩

4.2.2 sup

lemma *Err_sup_Err [simp]:*
"Err +_(Err.sup f) x = Err"
⟨*proof*⟩

lemma *Err_sup_Err2 [simp]:*
"x +_(Err.sup f) Err = Err"
⟨*proof*⟩

lemma *Err_sup_OK [simp]:*
"OK x +_(Err.sup f) OK y = OK(x +_f y)"
⟨*proof*⟩

lemma *Err_sup_eq_OK_conv [iff]:*
"(Err.sup f ex ey = OK z) = (∃x y. ex = OK x & ey = OK y & f x y = z)"
⟨*proof*⟩

lemma *Err_sup_eq_Err [iff]:*
"(Err.sup f ex ey = Err) = (ex=Err | ey=Err)"
⟨*proof*⟩

4.2.3 semilat (err A) (le r) f

lemma *semilat_le_err_Err_plus [simp]:*
"⟦ x ∈ err A; semilat(err A, le r, f) ⟧ ⟹ Err +_f x = Err"
⟨*proof*⟩

lemma *semilat_le_err_plus_Err [simp]:*
"⟦ x ∈ err A; semilat(err A, le r, f) ⟧ ⟹ x +_f Err = Err"
⟨*proof*⟩

lemma *semilat_le_err_OK1:*
"⟦ x ∈ A; y ∈ A; semilat(err A, le r, f); OK x +_f OK y = OK z ⟧
⟹ x <=_r z"
⟨*proof*⟩

lemma *semilat_le_err_OK2:*
"⟦ x ∈ A; y ∈ A; semilat(err A, le r, f); OK x +_f OK y = OK z ⟧
⟹ y <=_r z"
⟨*proof*⟩

lemma *eq_order_le:*
"⟦ x=y; order r ⟧ ⟹ x <=_r y"

⟨*proof*⟩

lemma *OK_plus_OK_eq_Err_conv [simp]:*
assumes *"x ∈ A"* **and** *"y ∈ A"* **and** *"semilat(err A, le r, fe)"*
shows *"((OK x) +_fe (OK y) = Err) = (¬(∃z∈A. x <=_r z & y <=_r z))"*
⟨*proof*⟩

4.2.4 semilat (err (Union AS))

lemma *all_bex_swap_lemma [iff]:*
"(∀x. (∃y∈A. x = f y) ⟶ P x) = (∀y∈A. P(f y))"
⟨*proof*⟩

lemma *closed_err_Union_lift2I:*
"⟦ ∀A∈AS. closed (err A) (lift2 f); AS ≠ {};
∀A∈AS. ∀B∈AS. A≠B ⟶ (∀a∈A. ∀b∈B. a +_f b = Err) ⟧
⟹ closed (err (⋃AS)) (lift2 f)"
⟨*proof*⟩

If *AS = {}* the thm collapses to *order r ∧ closed {Err} f ∧ Err ⊔$_f$ Err = Err* which may not hold

lemma *err_semilat_UnionI:*
"⟦ ∀A∈AS. err_semilat(A, r, f); AS ≠ {};
∀A∈AS. ∀B∈AS. A≠B ⟶ (∀a∈A. ∀b∈B. ¬ a <=_r b & a +_f b = Err) ⟧
⟹ err_semilat (⋃AS, r, f)"
⟨*proof*⟩

end

4.3 Fixed Length Lists

theory *Listn*
imports *Err*
begin

definition *list :: "nat ⇒ 'a set ⇒ 'a list set"* **where**
"list n A == {xs. length xs = n & set xs <= A}"

definition *le :: "'a ord ⇒ ('a list)ord"* **where**
"le r == list_all2 (%x y. x <=_r y)"

abbreviation
lesublist_syntax :: "'a list ⇒ 'a ord ⇒ 'a list ⇒ bool"
(‹(_ /<=[_] _)› [50, 0, 51] 50)
where *"x <=[r] y == x <=_(le r) y"*

abbreviation
lesssublist_syntax :: "'a list ⇒ 'a ord ⇒ 'a list ⇒ bool"
(‹(_ /<[_] _)› [50, 0, 51] 50)
where *"x <[r] y == x <_(le r) y"*

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

abbreviation
```
  plussublist_syntax :: "'a list ⇒ ('a ⇒ 'b ⇒ 'c) ⇒ 'b list ⇒ 'c list"
       (‹(_ /+[_] _)› [65, 0, 66] 65)
  where "x +[f] y == x +_(map2 f) y"
```

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

definition `sl :: "nat ⇒ 'a sl ⇒ 'a list sl"` **where**
```
"sl n == %(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 == %xs ys. if size xs = size ys then coalesce(xs +[f] ys) else Err"
```

definition `upto_esl :: "nat ⇒ 'a esl ⇒ 'a list esl"` **where**
```
"upto_esl m == %(A,r,f). (⋃{list n A |n. n <= m}, le r, sup f)"
```

lemmas `[simp] = set_update_subsetI`

lemma `unfold_lesub_list:`
```
  "xs <=[r] ys == Listn.le r xs ys"
```
⟨*proof*⟩

lemma `Nil_le_conv [iff]:`
```
  "([] <=[r] ys) = (ys = [])"
```
⟨*proof*⟩

lemma `Cons_notle_Nil [iff]:`
```
  "~ x#xs <=[r] []"
```
⟨*proof*⟩

lemma `Cons_le_Cons [iff]:`
```
  "x#xs <=[r] y#ys = (x <=_r y & xs <=[r] ys)"
```
⟨*proof*⟩

lemma `Cons_less_Conss [simp]:`
```
  "order r ⟹
  x#xs <_(Listn.le r) y#ys =
  (x <_r y & xs <=[r] ys  |  x = y & xs <_(Listn.le r) ys)"
```
⟨*proof*⟩

lemma `list_update_le_cong:`
```
  "⟦ i<size xs; xs <=[r] ys; x <=_r y ⟧ ⟹ xs[i:=x] <=[r] ys[i:=y]"
```
⟨*proof*⟩

lemma `le_listD:`
```
  "⟦ xs <=[r] ys; p < size xs ⟧ ⟹ xs!p <=_r ys!p"
```
⟨*proof*⟩

lemma `le_list_refl:`

```
  "∀x. x <=_r x ⟹ xs <=[r] xs"
```
⟨*proof*⟩

lemma *le_list_trans:*
```
  "⟦ order r; xs <=[r] ys; ys <=[r] zs ⟧ ⟹ xs <=[r] zs"
```
⟨*proof*⟩

lemma *le_list_antisym:*
```
  "⟦ order r; xs <=[r] ys; ys <=[r] xs ⟧ ⟹ xs = ys"
```
⟨*proof*⟩

lemma *order_listI [simp, intro!]:*
```
  "order r ⟹ order(Listn.le r)"
```
⟨*proof*⟩

lemma *lesub_list_impl_same_size [simp]:*
```
  "xs <=[r] ys ⟹ size ys = size xs"
```
⟨*proof*⟩

lemma *lesssub_list_impl_same_size:*
```
  "xs <_(Listn.le r) ys ⟹ size ys = size xs"
```
⟨*proof*⟩

lemma *le_list_appendI:*
```
  "⋀b c d. a <=[r] b ⟹ c <=[r] d ⟹ a@c <=[r] b@d"
```
⟨*proof*⟩

lemma *le_listI:*
```
  "length a = length b ⟹ (⋀n. n < length a ⟹ a!n <=_r b!n) ⟹ a <=[r] b"
```
⟨*proof*⟩

lemma *listI:*
```
  "⟦ length xs = n; set xs <= A ⟧ ⟹ xs ∈ list n A"
```
⟨*proof*⟩

lemma *listE_length [simp]:*
```
   "xs ∈ list n A ⟹ length xs = n"
```
⟨*proof*⟩

lemma *less_lengthI:*
```
  "⟦ xs ∈ list n A; p < n ⟧ ⟹ p < length xs"
```
⟨*proof*⟩

lemma *listE_set [simp]:*
```
  "xs ∈ list n A ⟹ set xs <= A"
```
⟨*proof*⟩

lemma *list_0 [simp]:*
```
  "list 0 A = {[]}"
```
⟨*proof*⟩

lemma *in_list_Suc_iff:*
```
  "(xs ∈ list (Suc n) A) = (∃y∈ A. ∃ys∈ list n A. xs = y#ys)"
```

⟨*proof*⟩

lemma *Cons_in_list_Suc [iff]:*
"(x#xs ∈ list (Suc n) A) = (x∈ A & xs ∈ list n A)"
⟨*proof*⟩

lemma *list_not_empty:*
"∃a. a∈ A ⟹ ∃xs. xs ∈ list n A"
⟨*proof*⟩

lemma *nth_in [rule_format, simp]:*
"∀i n. length xs = n ⟶ set xs <= A ⟶ i < n ⟶ (xs!i) ∈ A"
⟨*proof*⟩

lemma *listE_nth_in:*
"⟦ xs ∈ list n A; i < n ⟧ ⟹ (xs!i) ∈ A"
⟨*proof*⟩

lemma *listn_Cons_Suc [elim!]:*
"l#xs ∈ list n A ⟹ (⋀n'. n = Suc n' ⟹ l ∈ A ⟹ xs ∈ list n' A ⟹ P) ⟹ P"
⟨*proof*⟩

lemma *listn_appendE [elim!]:*
"a@b ∈ list n A ⟹ (⋀n1 n2. n=n1+n2 ⟹ a ∈ list n1 A ⟹ b ∈ list n2 A ⟹ P) ⟹ P"
⟨*proof*⟩

lemma *listt_update_in_list [simp, intro!]:*
"⟦ xs ∈ list n A; x∈ A ⟧ ⟹ xs[i := x] ∈ list n A"
⟨*proof*⟩

lemma *plus_list_Nil [simp]:*
"[] +[f] xs = []"
⟨*proof*⟩

lemma *plus_list_Cons [simp]:*
"(x#xs) +[f] ys = (case ys of [] ⇒ [] | y#ys ⇒ (x +_f y)#(xs +[f] ys))"
⟨*proof*⟩

lemma *length_plus_list [rule_format, simp]:*
"∀ys. length(xs +[f] ys) = min(length xs) (length ys)"
⟨*proof*⟩

lemma *nth_plus_list [rule_format, simp]:*
"∀xs ys i. length xs = n ⟶ length ys = n ⟶ i<n ⟶
(xs +[f] ys)!i = (xs!i) +_f (ys!i)"
⟨*proof*⟩

lemma (**in** *Semilat*) *plus_list_ub1 [rule_format]:*
"⟦ set xs <= A; set ys <= A; size xs = size ys ⟧

```
  ⟹ xs <=[r] xs +[f] ys"
```
⟨*proof*⟩

lemma (**in** *Semilat*) *plus_list_ub2:*
```
 "⟦set xs <= A; set ys <= A; size xs = size ys ⟧
  ⟹ ys <=[r] xs +[f] ys"
```
⟨*proof*⟩

lemma (**in** *Semilat*) *plus_list_lub [rule_format]:*
shows "∀ *xs ys zs. set xs <= A ⟶ set ys <= A ⟶ set zs <= A*
```
  ⟶ size xs = n & size ys = n ⟶
  xs <=[r] zs & ys <=[r] zs ⟶ xs +[f] ys <=[r] zs"
```
⟨*proof*⟩

lemma (**in** *Semilat*) *list_update_incr [rule_format]:*
```
 "x∈ A ⟹ set xs <= A ⟶
  (∀ i. i<size xs ⟶ xs <=[r] xs[i := x +_f xs!i])"
```
⟨*proof*⟩

lemma *acc_le_listI [intro!]:*
```
  "⟦ order r; acc r ⟧ ⟹ acc(Listn.le r)"
```
⟨*proof*⟩

lemma *closed_listI:*
```
  "closed S f ⟹ closed (list n S) (map2 f)"
```
⟨*proof*⟩

lemma *Listn_sl_aux:*
assumes "*semilat (A, r, f)*" **shows** "*semilat (Listn.sl n (A,r,f))*"
⟨*proof*⟩

lemma *Listn_sl:* "⋀*L. semilat L ⟹ semilat (Listn.sl n L)*"
⟨*proof*⟩

lemma *coalesce_in_err_list [rule_format]:*
```
  "∀xes. xes ∈ list n (err A) ⟶ coalesce xes ∈ err(list n A)"
```
⟨*proof*⟩

lemma *lem:* "⋀*x xs. x +_(#) xs = x#xs*"
⟨*proof*⟩

lemma *coalesce_eq_OK1_D [rule_format]:*
```
  "semilat(err A, Err.le r, lift2 f) ⟹
  ∀xs. xs ∈ list n A ⟶ (∀ys. ys ∈ list n A ⟶
  (∀zs. coalesce (xs +[f] ys) = OK zs ⟶ xs <=[r] zs))"
```
⟨*proof*⟩

lemma *coalesce_eq_OK2_D [rule_format]:*
```
  "semilat(err A, Err.le r, lift2 f) ⟹
  ∀xs. xs ∈ list n A ⟶ (∀ys. ys ∈ list n A ⟶
  (∀zs. coalesce (xs +[f] ys) = OK zs ⟶ ys <=[r] zs))"
```
⟨*proof*⟩

lemma *lift2_le_ub:*
```
"⟦ semilat(err A, Err.le r, lift2 f); x∈ A; y∈ A; x +_f y = OK z;
    u∈ A; x <=_r u; y <=_r u ⟧ ⟹ z <=_r u"
```
⟨*proof*⟩

lemma *coalesce_eq_OK_ub_D [rule_format]:*
```
"semilat(err A, Err.le r, lift2 f) ⟹
∀xs. xs ∈ list n A ⟶ (∀ys. ys ∈ list n A ⟶
(∀zs us. coalesce (xs +[f] ys) = OK zs ∧ xs <=[r] us ∧ ys <=[r] us
        ∧ us ∈ list n A ⟶ zs <=[r] us))"
```
⟨*proof*⟩

lemma *lift2_eq_ErrD:*
```
"⟦ x +_f y = Err; semilat(err A, Err.le r, lift2 f); x∈ A; y∈ A ⟧
⟹ ~(∃u∈ A. x <=_r u & y <=_r u)"
```
⟨*proof*⟩

lemma *coalesce_eq_Err_D [rule_format]:*
```
"⟦ semilat(err A, Err.le r, lift2 f) ⟧
⟹ ∀xs. xs ∈ list n A ⟶ (∀ys. ys ∈ list n A ⟶
    coalesce (xs +[f] ys) = Err ⟶
    ¬(∃zs∈ list n A. xs <=[r] zs ∧ ys <=[r] zs))"
```
⟨*proof*⟩

lemma *closed_err_lift2_conv:*
```
"closed (err A) (lift2 f) = (∀x∈ A. ∀y∈ A. x +_f y ∈ err A)"
```
⟨*proof*⟩

lemma *closed_map2_list [rule_format]:*
```
"closed (err A) (lift2 f) ⟹
∀xs. xs ∈ list n A ⟶ (∀ys. ys ∈ list n A ⟶
map2 f xs ys ∈ list n (err A))"
```
⟨*proof*⟩

lemma *closed_lift2_sup:*
```
"closed (err A) (lift2 f) ⟹
closed (err (list n A)) (lift2 (sup f))"
```
⟨*proof*⟩

lemma *err_semilat_sup:*
```
"err_semilat (A,r,f) ⟹
err_semilat (list n A, Listn.le r, sup f)"
```
⟨*proof*⟩

lemma *err_semilat_upto_esl:*
```
"⋀L. err_semilat L ⟹ err_semilat(upto_esl m L)"
```
⟨*proof*⟩

end

4.4 Typing and Dataflow Analysis Framework

theory *Typing_Framework*
imports *Listn*
begin

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

type_synonym *'s step_type = "nat ⇒ 's ⇒ (nat × 's) list"*

definition *stable :: "'s ord ⇒ 's step_type ⇒ 's list ⇒ nat ⇒ bool"* **where**
"stable r step ss p == ∀(q,s')∈set(step p (ss!p)). s' <=_r ss!q"

definition *stables :: "'s ord ⇒ 's step_type ⇒ 's list ⇒ bool"* **where**
"stables r step ss == ∀p<size ss. stable r step ss p"

definition *wt_step ::*
"'s ord ⇒ 's ⇒ 's step_type ⇒ 's list ⇒ bool" **where**
"wt_step r T step ts ==
∀p<size(ts). ts!p ~= T & stable r step ts p"

definition *is_bcv :: "'s ord ⇒ 's ⇒ 's step_type*
⇒ nat ⇒ 's set ⇒ ('s list ⇒ 's list) ⇒ bool" **where**
"is_bcv r T step n A bcv == ∀ss ∈ list n A.
(∀p<n. (bcv ss)!p ~= T) =
(∃ts ∈ list n A. ss <=[r] ts & wt_step r T step ts)"

end

4.5 Products as Semilattices

theory *Product*
imports *Err*
begin

definition *le :: "'a ord ⇒ 'b ord ⇒ ('a * 'b) ord"* **where**
"le rA rB == %(a,b) (a',b'). a <=_rA a' & b <=_rB b'"

definition *sup :: "'a ebinop ⇒ 'b ebinop ⇒ ('a * 'b)ebinop"* **where**
"sup f g == %(a1,b1)(a2,b2). Err.sup Pair (a1 +_f a2) (b1 +_g b2)"

definition *esl :: "'a esl ⇒ 'b esl ⇒ ('a * 'b) esl"* **where**
"esl == %(A,rA,fA) (B,rB,fB). (A × B, le rA rB, sup fA fB)"

abbreviation
*lesubprod_sntax :: "'a * 'b ⇒ 'a ord ⇒ 'b ord ⇒ 'a * 'b ⇒ bool"*
(‹(_ /<='(_,_') _)› [50, 0, 0, 51] 50)
where *"p <=(rA,rB) q == p <=_(le rA rB) q"*

lemma *unfold_lesub_prod:*
"p <=(rA,rB) q == le rA rB p q"
⟨*proof*⟩

lemma *le_prod_Pair_conv [iff]:*
"((a1,b1) <=(rA,rB) (a2,b2)) = (a1 <=_rA a2 & b1 <=_rB b2)"

⟨*proof*⟩

lemma *less_prod_Pair_conv:*
"((a1,b1) <_(Product.le rA rB) (a2,b2)) =
(a1 <_rA a2 & b1 <=_rB b2 | a1 <=_rA a2 & b1 <_rB b2)"
⟨*proof*⟩

lemma *order_le_prod [iff]:*
"order(Product.le rA rB) = (order rA & order rB)"
⟨*proof*⟩

lemma *acc_le_prodI [intro!]:*
"⟦ acc r_A*; acc* r_B *⟧ ⟹ acc(Product.le* r_A r_B*)"*
⟨*proof*⟩

lemma *closed_lift2_sup:*
"⟦ closed (err A) (lift2 f); closed (err B) (lift2 g) ⟧ ⟹
closed (err(A×B)) (lift2(sup f g))"
⟨*proof*⟩

lemma *unfold_plussub_lift2:*
"e1 +_(lift2 f) e2 == lift2 f e1 e2"
⟨*proof*⟩

lemma *plus_eq_Err_conv [simp]:*
assumes *"x ∈ A"* **and** *"y ∈ A"*
and *"semilat(err A, Err.le r, lift2 f)"*
shows *"(x +_f y = Err) = (¬(∃z∈A. x <=_r z & y <=_r z))"*
⟨*proof*⟩

lemma *err_semilat_Product_esl:*
"⋀L1 L2. ⟦ err_semilat L1; err_semilat L2 ⟧ ⟹ err_semilat(Product.esl L1 L2)"
⟨*proof*⟩

end

4.6 More on Semilattices

theory *SemilatAlg*
imports *Typing_Framework Product*
begin

definition *lesubstep_type :: "(nat × 's) list ⇒ 's ord ⇒ (nat × 's) list ⇒ bool"*
(⟨(_ /≤|_| _)⟩ [50, 0, 51] 50) **where**
"x ≤|r| y ≡ ∀(p,s) ∈ set x. ∃s'. (p,s') ∈ set y ∧ s <=_r s'"

primrec *plusplussub :: "'a list ⇒ ('a ⇒ 'a ⇒ 'a) ⇒ 'a ⇒ 'a" (⟨(_ /++'__ _)⟩ [65, 1000, 66] 65)* **where**
"[] ++_f y = y"
| "(x#xs) ++_f y = xs ++_f (x +_f y)"

definition *bounded :: "'s step_type ⇒ nat ⇒ bool"* **where**

"bounded step n == ∀p<n. ∀s. ∀(q,t)∈set(step p s). q<n"

definition *pres_type :: "'s step_type ⇒ nat ⇒ 's set ⇒ bool"* **where**
"pres_type step n A == ∀s∈A. ∀p<n. ∀(q,s')∈set (step p s). s' ∈ A"

definition *mono :: "'s ord ⇒ 's step_type ⇒ nat ⇒ 's set ⇒ bool"* **where**
"mono r step n A ==
 ∀s p t. s ∈ A ∧ p < n ∧ s <=_r t ⟶ step p s ≤|r| step p t"

lemma *pres_typeD:*
"⟦ pres_type step n A; s∈A; p<n; (q,s')∈set (step p s) ⟧ ⟹ s' ∈ A"
⟨*proof*⟩

lemma *monoD:*
"⟦ mono r step n A; p < n; s∈A; s <=_r t ⟧ ⟹ step p s ≤|r| step p t"
⟨*proof*⟩

lemma *boundedD:*
"⟦ bounded step n; p < n; (q,t) ∈ set (step p xs) ⟧ ⟹ q < n"
⟨*proof*⟩

lemma *lesubstep_type_refl [simp, intro]:*
"(⋀x. x <=_r x) ⟹ x ≤|r| x"
⟨*proof*⟩

lemma *lesub_step_typeD:*
"a ≤|r| b ⟹ (x,y) ∈ set a ⟹ ∃y'. (x, y') ∈ set b ∧ y <=_r y'"
⟨*proof*⟩

lemma *list_update_le_listI [rule_format]:*
"set xs <= A ⟶ set ys <= A ⟶ xs <=[r] ys ⟶ p < size xs ⟶
 x <=_r ys!p ⟶ semilat(A,r,f) ⟶ x∈A ⟶
 xs[p := x +_f xs!p] <=[r] ys"
⟨*proof*⟩

lemma *plusplus_closed:* **assumes** *"semilat (A, r, f)"* **shows**
"⋀y. ⟦ set x ⊆ A; y ∈ A⟧ ⟹ x ++_f y ∈ A" (**is** *"PROP ?P"*)
⟨*proof*⟩

lemma (**in** *Semilat*) *pp_ub2:*
"⋀y. ⟦ set x ⊆ A; y ∈ A⟧ ⟹ y <=_r x ++_f y"
⟨*proof*⟩

lemma (**in** *Semilat*) *pp_ub1:*
shows "⋀y. ⟦set ls ⊆ A; y ∈ A; x ∈ set ls⟧ ⟹ x <=_r ls ++_f y"
⟨*proof*⟩

lemma (**in** *Semilat*) *pp_lub:*
assumes *z:* "z ∈ A"
shows

```
  "⋀y. y ∈ A ⟹ set xs ⊆ A ⟹ ∀x ∈ set xs. x <=_r z ⟹ y <=_r z ⟹ xs ++_f y <=_r
z"
```
⟨*proof*⟩

lemma *ub1':*
```
  assumes "semilat (A, r, f)"
  shows "⟦∀(p,s) ∈ set S. s ∈ A; y ∈ A; (a,b) ∈ set S⟧
  ⟹ b <=_r map snd [(p', t')←S. p' = a] ++_f y"
```
⟨*proof*⟩

lemma *plusplus_empty:*
```
  "∀s'. (q, s') ∈ set S ⟶ s' +_f ss ! q = ss ! q ⟹
   (map snd [(p', t') ← S. p' = q] ++_f ss ! q) = ss ! q"
```
⟨*proof*⟩

end

4.7 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 ts ≡ wt_step (Err.le r) Err step ts"

definition *wt_app_eff :: "'s ord ⇒ (nat ⇒ 's ⇒ bool) ⇒ 's step_type ⇒ 's list ⇒ bool"*
where
```
"wt_app_eff r app step ts ≡
  ∀p < size ts. app p (ts!p) ∧ (∀(q,t) ∈ set (step p (ts!p)). t <=_r ts!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 t' ⇒ if app p t' then map_snd OK (step p t') 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"
```
⟨*proof*⟩

lemma *in_map_sndD:* `"(a,b) ∈ set (map_snd f xs) ⟹ ∃b'. (a,b') ∈ set xs"`
⟨*proof*⟩

lemma *bounded_err_stepI:*
```
"∀p. p < n ⟶ (∀s. ap p s ⟶ (∀(q,s') ∈ set (step p s). q < n))
⟹ bounded (err_step n ap step) n"
```
⟨*proof*⟩

lemma *bounded_lift:*
```
"bounded step n ⟹ bounded (err_step n app step) n"
```
⟨*proof*⟩

lemma *le_list_map_OK [simp]:*
```
"⋀b. map OK a <=[Err.le r] map OK b = (a <=[r] b)"
```
⟨*proof*⟩

lemma *map_snd_lessI:*
```
"x ≤|r| y ⟹ map_snd OK x ≤|Err.le r| map_snd OK y"
```
⟨*proof*⟩

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 ⟶ step p s ≤|r| step p t ⟹
mono (Err.le r) (err_step n app step) n (err A)"
```
⟨*proof*⟩

lemma *in_errorD:*
```
"(x,y) ∈ set (error n) ⟹ y = Err"
```
⟨*proof*⟩

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)"
```
⟨*proof*⟩

There used to be a condition here that each instruction must have a successor. This is not needed any more, because the definition of *error* trivially ensures that there is a successor for the critical case where *app* does not hold.

lemma *wt_err_imp_wt_app_eff:*
assumes *wt:* `"wt_err_step r (err_step (size ts) app step) ts"`
assumes *b:* `"bounded (err_step (size ts) app step) (size ts)"`

shows *"wt_app_eff r app step (map ok_val ts)"*
⟨*proof*⟩

lemma *wt_app_eff_imp_wt_err:*
assumes *app_eff: "wt_app_eff r app step ts"*
assumes *bounded: "bounded (err_step (size ts) app step) (size ts)"*
shows *"wt_err_step r (err_step (size ts) app step) (map OK ts)"*
⟨*proof*⟩

end

4.8 Kildall's Algorithm

theory *Kildall*
imports *SemilatAlg "HOL-Library.While_Combinator"*
begin

primrec *propa :: "'s binop ⇒ (nat × 's) list ⇒ 's list ⇒ nat set ⇒ 's list * nat set"* **where**
"propa f []   ss w = (ss,w)"
| "propa f (q'#qs) ss w = (let (q,t) = q';
u = t +_f ss!q;
w' = (if u = ss!q then w else insert q w)
in propa f qs (ss[q := u]) w')"

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

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

definition *kildall :: "'s ord ⇒ 's binop ⇒ 's step_type ⇒ 's list ⇒ 's list"* **where**
"kildall r f step ss == fst(iter f step ss (unstables r step ss))"

primrec *merges :: "'s binop ⇒ (nat × 's) list ⇒ 's list ⇒ 's list"* **where**
"merges f []   ss = ss"
| "merges f (p'#ps) ss = (let (p,s) = p' in merges f ps (ss[p := s +_f ss!p]))"

lemmas *[simp] = Let_def Semilat.le_iff_plus_unchanged [OF Semilat.intro, symmetric]*

lemma **(in** *Semilat)* *nth_merges:*
"⋀ss. ⟦p < length ss; ss ∈ list n A; ∀(p,t)∈set ps. p<n ∧ t∈A ⟧ ⟹
(merges f ps ss)!p = map snd [(p',t') ← ps. p'=p] ++_f ss!p"
(is *"⋀ss. ⟦_; _; ?steptype ps⟧ ⟹ ?P ss ps")*
⟨*proof*⟩

lemma *length_merges [simp]: "size(merges f ps ss) = size ss"*
⟨*proof*⟩

lemma (**in** *Semilat*) *merges_preserves_type_lemma:*
shows *"∀xs. xs ∈ list n A ⟶ (∀(p,x) ∈ set ps. p<n ∧ x∈A)*
⟶ merges f ps xs ∈ list n A"
⟨*proof*⟩

lemma (**in** *Semilat*) *merges_preserves_type [simp]:*
"⟦ xs ∈ list n A; ∀(p,x) ∈ set ps. p<n ∧ x∈A ⟧
⟹ merges f ps xs ∈ list n A"
⟨*proof*⟩

lemma (**in** *Semilat*) *merges_incr_lemma:*
"∀xs. xs ∈ list n A ⟶ (∀(p,x)∈set ps. p<size xs ∧ x ∈ A) ⟶ xs <=[r] merges f ps xs"
⟨*proof*⟩

lemma (**in** *Semilat*) *merges_incr:*
"⟦ xs ∈ list n A; ∀(p,x)∈set ps. p<size xs ∧ x ∈ A ⟧
⟹ xs <=[r] merges f ps xs"
⟨*proof*⟩

lemma (**in** *Semilat*) *merges_same_conv [rule_format]:*
"(∀xs. xs ∈ list n A ⟶ (∀(p,x)∈set ps. p<size xs ∧ x∈A) ⟶
(merges f ps xs = xs) = (∀(p,x)∈set ps. x <=_r xs!p))"
⟨*proof*⟩

lemma (**in** *Semilat*) *list_update_le_listI [rule_format]:*
"set xs <= A ⟶ set ys <= A ⟶ xs <=[r] ys ⟶ p < size xs ⟶
x <=_r ys!p ⟶ x∈A ⟶ xs[p := x +_f xs!p] <=[r] ys"
⟨*proof*⟩

lemma (**in** *Semilat*) *merges_pres_le_ub:*
assumes *"set ts <= A"* **and** *"set ss <= A"*
and *"∀(p,t)∈set ps. t <=_r ts!p ∧ t ∈ A ∧ p < size ts"* **and** *"ss <=[r] ts"*
shows *"merges f ps ss <=[r] ts"*
⟨*proof*⟩

lemma *decomp_propa:*
"⋀ss w. (∀(q,t)∈set qs. q < size ss) ⟹
propa f qs ss w =
(merges f qs ss, {q. ∃t. (q,t)∈set qs ∧ t +_f ss!q ≠ ss!q} Un w)"
⟨*proof*⟩

lemma (**in** *Semilat*) *stable_pres_lemma:*
shows "⟦*pres_type step n A; bounded step n;*
ss ∈ list n A; p ∈ w; ∀q∈w. q < n;
∀q. q < n ⟶ q ∉ w ⟶ stable r step ss q; q < n;
∀s'. (q,s') ∈ set (step p (ss ! p)) ⟶ s' +_f ss ! q = ss ! q;
q ∉ w ∨ q = p ⟧
⟹ stable r step (merges f (step p (ss!p)) ss) q"
⟨*proof*⟩

lemma (**in** *Semilat*) *merges_bounded_lemma:*
"⟦ *mono r step n A; bounded step n;*
∀(p',s') ∈ set (step p (ss!p)). s' ∈ A; ss ∈ list n A; ts ∈ list n A; p < n;
ss <=[r] ts; ∀p. p < n ⟶ stable r step ts p ⟧
⟹ merges f (step p (ss!p)) ss <=[r] ts"
⟨*proof*⟩

lemma *termination_lemma:*
assumes *semilat: "semilat (A, r, f)"*
shows "⟦ *ss ∈ list n A; ∀(q,t)∈set qs. q<n ∧ t∈A; p∈w* ⟧ *⟹*
ss <[r] merges f qs ss ∨
merges f qs ss = ss ∧ {q. ∃t. (q,t)∈set qs ∧ t +_f ss!q ≠ ss!q} Un (w-{p}) < w" (**is** *"PROP ?P"*)
⟨*proof*⟩

lemma *iter_properties[rule_format]:*
assumes *semilat: "semilat (A, r, f)"*
shows "⟦ *acc r ; pres_type step n A; mono r step n A;*
bounded step n; ∀p∈w0. p < n; ss0 ∈ list n A;
∀p<n. p ∉ w0 ⟶ stable r step ss0 p ⟧ *⟹*
iter f step ss0 w0 = (ss',w')
⟶
ss' ∈ list n A ∧ stables r step ss' ∧ ss0 <=[r] ss' ∧
(∀ts∈list n A. ss0 <=[r] ts ∧ stables r step ts ⟶ ss' <=[r] ts)"
(**is** *"PROP ?P"*)
⟨*proof*⟩

lemma *kildall_properties:*
assumes *semilat: "semilat (A, r, f)"*
shows "⟦ *acc r; pres_type step n A; mono r step n A;*
bounded step n; ss0 ∈ list n A ⟧ *⟹*
kildall r f step ss0 ∈ list n A ∧
stables r step (kildall r f step ss0) ∧
ss0 <=[r] kildall r f step ss0 ∧
(∀ts∈list n A. ss0 <=[r] ts ∧ stables r step ts ⟶
kildall r f step ss0 <=[r] ts)"
(**is** *"PROP ?P"*)
⟨*proof*⟩

lemma *is_bcv_kildall:*
assumes *semilat: "semilat (A, r, f)"*

shows "⟦ *acc r; top r T; pres_type step n A; bounded step n; mono r step n A* ⟧
  ⟹ *is_bcv r T step n A (kildall r f step)*"
  (**is** "*PROP ?P*")
⟨*proof*⟩

end

4.9 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 <=_r y)"

definition *opt :: "'a set ⇒ 'a option set"* **where**
"opt A == insert None {x. ∃y∈A. x = Some y}"

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 == %(A,r,f). (opt A, le r, sup f)"

lemma *unfold_le_opt:*
  "o1 <=_(le r) o2 =
  (case o2 of None ⇒ o1=None |
  Some y ⇒ (case o1 of None ⇒ True | Some x ⇒ x <=_r y))"
⟨*proof*⟩

lemma *le_opt_refl:*
  "order r ⟹ o1 <=_(le r) o1"
⟨*proof*⟩

lemma *le_opt_trans [rule_format]:*
  "order r ⟹
  o1 <=_(le r) o2 ⟶ o2 <=_(le r) o3 ⟶ o1 <=_(le r) o3"
⟨*proof*⟩

lemma *le_opt_antisym [rule_format]:*
  "order r ⟹ o1 <=_(le r) o2 ⟶ o2 <=_(le r) o1 ⟶ o1=o2"
⟨*proof*⟩

lemma *order_le_opt [intro!,simp]:*
  "order r ⟹ order(le r)"
⟨*proof*⟩

lemma *None_bot [iff]:*

```
  "None <=_(le r) ox"
```
⟨*proof*⟩

lemma *Some_le [iff]:*
```
  "(Some x <=_(le r) ox) = (∃y. ox = Some y ∧ x <=_r y)"
```
⟨*proof*⟩

lemma *le_None [iff]:*
```
  "(ox <=_(le r) None) = (ox = None)"
```
⟨*proof*⟩

lemma *OK_None_bot [iff]:*
```
  "OK None <=_(Err.le (le r)) x"
```
⟨*proof*⟩

lemma *sup_None1 [iff]:*
```
  "x +_(sup f) None = OK x"
```
⟨*proof*⟩

lemma *sup_None2 [iff]:*
```
  "None +_(sup f) x = OK x"
```
⟨*proof*⟩

lemma *None_in_opt [iff]:*
```
  "None ∈ opt A"
```
⟨*proof*⟩

lemma *Some_in_opt [iff]:*
```
  "(Some x ∈ opt A) = (x∈A)"
```
⟨*proof*⟩

lemma *semilat_opt [intro, simp]:*
```
  "⋀L. err_semilat L ⟹ err_semilat (Opt.esl L)"
```
⟨*proof*⟩

lemma *top_le_opt_Some [iff]:*
```
  "top (le r) (Some T) = top r T"
```
⟨*proof*⟩

lemma *Top_le_conv:*
```
  "⟦ order r; top r T ⟧ ⟹ (T <=_r x) = (x = T)"
```
⟨*proof*⟩

lemma *acc_le_optI [intro!]:*
```
  "acc r ⟹ acc(le r)"
```
⟨*proof*⟩

lemma *option_map_in_optionI:*
```
  "⟦ ox ∈ opt S; ∀x∈S. ox = Some x ⟶ f x ∈ S ⟧
  ⟹ map_option f ox ∈ opt S"
```

⟨*proof*⟩

end

4.10 The Lightweight Bytecode Verifier

theory *LBVSpec*
imports *SemilatAlg Opt*
begin

type_synonym *'s certificate = "'s list"*

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

definition *wtl_inst :: "'s certificate ⇒ 's binop ⇒ 's ord ⇒ 's ⇒ 's step_type ⇒ nat ⇒ 's ⇒ 's"* **where**
```
"wtl_inst cert f r T step pc s ≡ merge cert f r T pc (step pc s) (cert!(pc+1))"
```

definition *wtl_cert :: "'s certificate ⇒ 's binop ⇒ 's ord ⇒ 's ⇒ 's ⇒ 's step_type ⇒ nat ⇒ 's ⇒ 's"* **where**
```
"wtl_cert cert f r T B step pc s ≡
  if cert!pc = B then
    wtl_inst cert f r T step pc s
  else
    if s <=_r (cert!pc) then wtl_inst cert f r T step pc (cert!pc) else T"
```

primrec *wtl_inst_list :: "'a list ⇒ 's certificate ⇒ 's binop ⇒ 's ord ⇒ 's ⇒ 's ⇒ 's step_type ⇒ nat ⇒ 's ⇒ 's"* **where**
```
  "wtl_inst_list []     cert f r T B step pc s = s"
| "wtl_inst_list (i#is) cert f r T B step pc s =
    (let s' = wtl_cert cert f r T B step pc s in
      if s' = T ∨ s = T then T else wtl_inst_list is cert f r T B step (pc+1) s')"
```

definition *cert_ok :: "'s certificate ⇒ nat ⇒ 's ⇒ 's ⇒ 's set ⇒ bool"* **where**
```
  "cert_ok cert n T B A ≡ (∀i < n. cert!i ∈ A ∧ cert!i ≠ T) ∧ (cert!n = B)"
```

definition *bottom :: "'a ord ⇒ 'a ⇒ bool"* **where**
```
  "bottom r B ≡ ∀x. B <=_r x"
```

locale *lbv = Semilat +*
```
  fixes T :: "'a" (<⊤>)
  fixes B :: "'a" (<⊥>)
  fixes step :: "'a step_type"
  assumes top: "top r ⊤"
  assumes T_A: "⊤ ∈ A"
  assumes bot: "bottom r ⊥"
```

assumes *B_A: "⊥ ∈ A"*

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

fixes *wti :: "'a certificate ⇒ nat ⇒ 'a ⇒ 'a"*
defines *wti_def: "wti cert ≡ wtl_inst cert f r ⊤ step"*

fixes *wtc :: "'a certificate ⇒ nat ⇒ 'a ⇒ 'a"*
defines *wtc_def: "wtc cert ≡ wtl_cert cert f r ⊤ ⊥ step"*

fixes *wtl :: "'b list ⇒ 'a certificate ⇒ nat ⇒ 'a ⇒ 'a"*
defines *wtl_def: "wtl ins cert ≡ wtl_inst_list ins cert f r ⊤ ⊥ step"*

lemma (**in** *lbv*) *wti:*
"wti c pc s ≡ merge c pc (step pc s) (c!(pc+1))"
⟨*proof*⟩

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 ⊤"
⟨*proof*⟩

lemma *cert_okD1 [intro?]:*
"cert_ok c n T B A ⟹ pc < n ⟹ c!pc ∈ A"
⟨*proof*⟩

lemma *cert_okD2 [intro?]:*
"cert_ok c n T B A ⟹ c!n = B"
⟨*proof*⟩

lemma *cert_okD3 [intro?]:*
"cert_ok c n T B A ⟹ B ∈ A ⟹ pc < n ⟹ c!Suc pc ∈ A"
⟨*proof*⟩

lemma *cert_okD4 [intro?]:*
"cert_ok c n T B A ⟹ pc < n ⟹ c!pc ≠ T"
⟨*proof*⟩

declare *Let_def [simp]*

4.10.1 more semilattice lemmas

lemma (**in** *lbv*) *sup_top [simp, elim]:*
assumes *x: "x ∈ A"*
shows *"x +_f ⊤ = ⊤"*
⟨*proof*⟩

lemma (**in** *lbv*) *plusplussup_top [simp, elim]:*
"set xs ⊆ A ⟹ xs ++_f ⊤ = ⊤"
⟨*proof*⟩

lemma (**in** *Semilat*) *pp_ub1':*
assumes *S:* "*snd‘set S* ⊆ *A*"
assumes *y:* "*y* ∈ *A*" **and** *ab:* "(*a*, *b*) ∈ *set S*"
shows "*b* <=_*r map snd* [(*p'*, *t'*) ← *S* . *p'* = *a*] ++_*f y*"
⟨*proof*⟩

lemma (**in** *lbv*) *bottom_le* [*simp*, *intro*]:
"⊥ <=_*r x*"
⟨*proof*⟩

lemma (**in** *lbv*) *le_bottom* [*simp*]:
"*x* <=_*r* ⊥ = (*x* = ⊥)"
⟨*proof*⟩

4.10.2 merge

lemma (**in** *lbv*) *merge_Nil* [*simp*]:
"*merge c pc* [] *x* = *x*" ⟨*proof*⟩

lemma (**in** *lbv*) *merge_Cons* [*simp*]:
"*merge c pc* (*l*#*ls*) *x* = *merge c pc ls* (*if fst l*=*pc*+1 *then snd l* +_*f x*
else if snd l <=_*r* (*c*!*fst l*) *then x*
else ⊤)"
⟨*proof*⟩

lemma (**in** *lbv*) *merge_Err* [*simp*]:
"*snd‘set ss* ⊆ *A* ⟹ *merge c pc ss* ⊤ = ⊤"
⟨*proof*⟩

lemma (**in** *lbv*) *merge_not_top:*
"⋀*x*. *snd‘set ss* ⊆ *A* ⟹ *merge c pc ss x* ≠ ⊤ ⟹
∀(*pc'*,*s'*) ∈ *set ss*. (*pc'* ≠ *pc*+1 ⟶ *s'* <=_*r* (*c*!*pc'*))"
(**is** "⋀*x*. *?set ss* ⟹ *?merge ss x* ⟹ *?P ss*")
⟨*proof*⟩

lemma (**in** *lbv*) *merge_def:*
shows
"⋀*x*. *x* ∈ *A* ⟹ *snd‘set ss* ⊆ *A* ⟹
merge c pc ss x =
(*if* ∀(*pc'*,*s'*) ∈ *set ss*. *pc'*≠*pc*+1 ⟶ *s'* <=_*r c*!*pc' then*
map snd [(*p'*,*t'*) ← *ss*. *p'*=*pc*+1] ++_*f x*
else ⊤)"
(**is** "⋀*x*. _ ⟹ _ ⟹ *?merge ss x* = *?if ss x*" **is** "⋀*x*. _ ⟹ _ ⟹ *?P ss x*")
⟨*proof*⟩

lemma (**in** *lbv*) *merge_not_top_s:*
assumes *x:* "*x* ∈ *A*" **and** *ss:* "*snd‘set ss* ⊆ *A*"
assumes *m:* "*merge c pc ss x* ≠ ⊤"
shows "*merge c pc ss x* = (*map snd* [(*p'*,*t'*) ← *ss*. *p'*=*pc*+1] ++_*f x*)"
⟨*proof*⟩

4.10.3 wtl-inst-list

lemmas *[iff] = not_Err_eq*

lemma (**in** *lbv*) *wtl_Nil [simp]: "wtl [] c pc s = s"*
⟨*proof*⟩

lemma (**in** *lbv*) *wtl_Cons [simp]:*
"wtl (i#is) c pc s =
(let s' = wtc c pc s in if s' = ⊤ ∨ s = ⊤ then ⊤ else wtl is c (pc+1) s')"
⟨*proof*⟩

lemma (**in** *lbv*) *wtl_Cons_not_top:*
"wtl (i#is) c pc s ≠ ⊤ =
(wtc c pc s ≠ ⊤ ∧ s ≠ T ∧ wtl is c (pc+1) (wtc c pc s) ≠ ⊤)"
⟨*proof*⟩

lemma (**in** *lbv*) *wtl_top [simp]: "wtl ls c pc ⊤ = ⊤"*
⟨*proof*⟩

lemma (**in** *lbv*) *wtl_not_top:*
"wtl ls c pc s ≠ ⊤ ⟹ s ≠ ⊤"
⟨*proof*⟩

lemma (**in** *lbv*) *wtl_append [simp]:*
"⋀pc s. wtl (a@b) c pc s = wtl b c (pc+length a) (wtl a c pc s)"
⟨*proof*⟩

lemma (**in** *lbv*) *wtl_take:*
"wtl is c pc s ≠ ⊤ ⟹ wtl (take pc' is) c pc s ≠ ⊤"
(**is** *"?wtl is ≠ _ ⟹ _"*)
⟨*proof*⟩

lemma *take_Suc:*
"∀n. n < length l ⟶ take (Suc n) l = (take n l)@[l!n]" (**is** *"?P l"*)
⟨*proof*⟩

lemma (**in** *lbv*) *wtl_Suc:*
assumes *suc: "pc+1 < length is"*
assumes *wtl: "wtl (take pc is) c 0 s ≠ ⊤"*
shows *"wtl (take (pc+1) is) c 0 s = wtc c pc (wtl (take pc is) c 0 s)"*
⟨*proof*⟩

lemma (**in** *lbv*) *wtl_all:*
assumes *all: "wtl is c 0 s ≠ ⊤"* (**is** *"?wtl is ≠ _"*)
assumes *pc: "pc < length is"*
shows *"wtc c pc (wtl (take pc is) c 0 s) ≠ ⊤"*
⟨*proof*⟩

4.10.4 preserves-type

lemma (**in** *lbv*) *merge_pres:*
assumes *s0: "snd`set ss ⊆ A"* **and** *x: "x ∈ A"*
shows *"merge c pc ss x ∈ A"*
⟨*proof*⟩

lemma *pres_typeD2:*
"pres_type step n A ⟹ s ∈ A ⟹ p < n ⟹ snd‘set (step p s) ⊆ A"
⟨*proof*⟩

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"*
⟨*proof*⟩

lemma (**in** *lbv*) *wtc_pres:*
assumes *pres: "pres_type step n A"*
assumes *cert: "c!pc ∈ A"* **and** *cert': "c!(pc+1) ∈ A"*
assumes *s: "s ∈ A"* **and** *pc: "pc < n"*
shows *"wtc c pc s ∈ A"*
⟨*proof*⟩

lemma (**in** *lbv*) *wtl_pres:*
assumes *pres: "pres_type step (length is) A"*
assumes *cert: "cert_ok c (length is) ⊤ ⊥ A"*
assumes *s: "s ∈ A"*
assumes *all: "wtl is c 0 s ≠ ⊤"*
shows *"pc < length is ⟹ wtl (take pc is) c 0 s ∈ A"*
(**is** *"?len pc ⟹ ?wtl pc ∈ A"*)
⟨*proof*⟩

end

4.11 Correctness of the LBV

theory *LBVCorrect*
imports *LBVSpec Typing_Framework*
begin

locale *lbvs = lbv +*
fixes *s0 :: 'a (⟨s_0⟩)*
fixes *c :: "'a list"*
fixes *ins :: "'b list"*
fixes *phi :: "'a list" (⟨φ⟩)*
defines *phi_def:*
"φ ≡ map (λpc. if c!pc = ⊥ then wtl (take pc ins) c 0 s0 else c!pc)
[0..<length ins]"

assumes *bounded: "bounded step (length ins)"*
assumes *cert: "cert_ok c (length ins) ⊤ ⊥ A"*
assumes *pres: "pres_type step (length ins) A"*

lemma (in *lbvs*) *phi_None [intro?]:*
"⟦ *pc < length ins; c!pc = ⊥* ⟧ ⟹ *φ ! pc = wtl (take pc ins) c 0 s0*"
⟨*proof*⟩

lemma (in *lbvs*) *phi_Some [intro?]:*
"⟦ *pc < length ins; c!pc ≠ ⊥* ⟧ ⟹ *φ ! pc = c ! pc*"
⟨*proof*⟩

lemma (in *lbvs*) *phi_len [simp]:*
"*length φ = length ins*"
⟨*proof*⟩

lemma (in *lbvs*) *wtl_suc_pc:*
assumes *all:* "*wtl ins c 0 s₀ ≠ ⊤*"
assumes *pc:* "*pc+1 < length ins*"
shows "*wtl (take (pc+1) ins) c 0 s0 ⊑ᵣ φ!(pc+1)*"
⟨*proof*⟩

lemma (in *lbvs*) *wtl_stable:*
assumes *wtl:* "*wtl ins c 0 s0 ≠ ⊤*"
assumes *s0:* "*s0 ∈ A*"
assumes *pc:* "*pc < length ins*"
shows "*stable r step φ pc*"
⟨*proof*⟩

lemma (in *lbvs*) *phi_not_top:*
assumes *wtl:* "*wtl ins c 0 s0 ≠ ⊤*"
assumes *pc:* "*pc < length ins*"
shows "*φ!pc ≠ ⊤*"
⟨*proof*⟩

lemma (in *lbvs*) *phi_in_A:*
assumes *wtl:* "*wtl ins c 0 s0 ≠ ⊤*"
assumes *s0:* "*s0 ∈ A*"
shows "*φ ∈ list (length ins) A*"
⟨*proof*⟩

lemma (in *lbvs*) *phi0:*
assumes *wtl:* "*wtl ins c 0 s0 ≠ ⊤*"
assumes *0:* "*0 < length ins*"
shows "*s0 <=_r φ!0*"
⟨*proof*⟩

theorem (in *lbvs*) *wtl_sound:*
assumes *wtl:* "*wtl ins c 0 s0 ≠ ⊤*"
assumes *s0:* "*s0 ∈ A*"
shows "*∃ts. wt_step r ⊤ step ts*"
⟨*proof*⟩

theorem (**in** *lbvs*) *wtl_sound_strong:*
assumes *wtl: "wtl ins c 0 s0 ≠ ⊤"*
assumes *s0: "s0 ∈ A"*
assumes *nz: "0 < length ins"*
shows *"∃ts ∈ list (length ins) A. wt_step r ⊤ step ts ∧ s0 <=_r ts!0"*
⟨*proof*⟩

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 phi pc' ⟷
(∃pc s'. pc' ≠ pc+1 ∧ pc < length phi ∧ (pc',s') ∈ set (step pc (phi!pc)))"

definition *make_cert :: "['s step_type, 's list, 's] ⇒ 's certificate"* **where**
"make_cert step phi B =
map (λpc. if is_target step phi pc then phi!pc else B) [0..<length phi] @ [B]"

lemma *[code]:*
"is_target step phi pc' =
list_ex (λpc. pc' ≠ pc+1 ∧ List.member (map fst (step pc (phi!pc))) pc') [0..<length phi]"
⟨*proof*⟩

locale *lbvc = lbv +*
fixes *phi :: "'a list" (<φ>)*
fixes *c :: "'a list"*
defines *cert_def: "c ≡ make_cert step φ ⊥"*

assumes *mono: "mono r step (length φ) A"*
assumes *pres: "pres_type step (length φ) A"*
assumes *phi: "∀pc < length φ. φ!pc ∈ A ∧ φ!pc ≠ ⊤"*
assumes *bounded: "bounded step (length φ)"*

assumes *B_neq_T: "⊥ ≠ ⊤"*

lemma (**in** *lbvc*) *cert: "cert_ok c (length φ) ⊤ ⊥ A"*
⟨*proof*⟩

lemmas *[simp del] = split_paired_Ex*

lemma (**in** *lbvc*) *cert_target [intro?]:*
"⟦ (pc',s') ∈ set (step pc (φ!pc));

```
    pc' ≠ pc+1; pc < length φ; pc' < length φ ⟧
  ⟹ c!pc' = φ!pc'"
```
⟨*proof*⟩

lemma (**in** *lbvc*) *cert_approx [intro?]*:
```
  "⟦ pc < length φ; c!pc ≠ ⊥ ⟧
  ⟹ c!pc = φ!pc"
```
⟨*proof*⟩

lemma (**in** *lbv*) *le_top [simp, intro]*:
```
  "x <=_r ⊤"
```
⟨*proof*⟩

lemma (**in** *lbv*) *merge_mono*:
assumes *less*: `"ss2 ≤|r| ss1"`
assumes *x*: `"x ∈ A"`
assumes *ss1*: `"snd`set ss1 ⊆ A"`
assumes *ss2*: `"snd`set ss2 ⊆ A"`
shows `"merge c pc ss2 x <=_r merge c pc ss1 x"` (**is** `"?s2 <=_r ?s1"`)
⟨*proof*⟩

lemma (**in** *lbvc*) *wti_mono*:
assumes *less*: `"s2 <=_r s1"`
assumes *pc*: `"pc < length φ"`
assumes *s1*: `"s1 ∈ A"`
assumes *s2*: `"s2 ∈ A"`
shows `"wti c pc s2 <=_r wti c pc s1"` (**is** `"?s2' <=_r ?s1'"`)
⟨*proof*⟩

lemma (**in** *lbvc*) *wtc_mono*:
assumes *less*: `"s2 <=_r s1"`
assumes *pc*: `"pc < length φ"`
assumes *s1*: `"s1 ∈ A"`
assumes *s2*: `"s2 ∈ A"`
shows `"wtc c pc s2 <=_r wtc c pc s1"` (**is** `"?s2' <=_r ?s1'"`)
⟨*proof*⟩

lemma (**in** *lbv*) *top_le_conv [simp]*:
```
  "⊤ <=_r x = (x = ⊤)"
```
⟨*proof*⟩

lemma (**in** *lbv*) *neq_top [simp, elim]*:
```
  "⟦ x <=_r y; y ≠ ⊤ ⟧ ⟹ x ≠ ⊤"
```
⟨*proof*⟩

lemma (**in** *lbvc*) *stable_wti*:
assumes *stable*: `"stable r step φ pc"`
assumes *pc*: `"pc < length φ"`

shows *"wti c pc (φ!pc) ≠ ⊤"*
⟨*proof*⟩

lemma (**in** *lbvc*) *wti_less:*
assumes *stable: "stable r step φ pc"*
assumes *suc_pc: "Suc pc < length φ"*
shows *"wti c pc (φ!pc) <=_r φ!Suc pc"* (**is** *"?wti <=_r _"*)
⟨*proof*⟩

lemma (**in** *lbvc*) *stable_wtc:*
assumes *stable: "stable r step phi pc"*
assumes *pc: "pc < length φ"*
shows *"wtc c pc (φ!pc) ≠ ⊤"*
⟨*proof*⟩

lemma (**in** *lbvc*) *wtc_less:*
assumes *stable: "stable r step φ pc"*
assumes *suc_pc: "Suc pc < length φ"*
shows *"wtc c pc (φ!pc) <=_r φ!Suc pc"* (**is** *"?wtc <=_r _"*)
⟨*proof*⟩

lemma (**in** *lbvc*) *wt_step_wtl_lemma:*
assumes *wt_step: "wt_step r ⊤ step φ"*
shows *"⋀pc s. pc+length ls = length φ ⟹ s <=_r φ!pc ⟹ s ∈ A ⟹ s≠⊤ ⟹*
wtl ls c pc s ≠ ⊤"
(**is** *"⋀pc s. _ ⟹ _ ⟹ _ ⟹ _ ⟹ ?wtl ls pc s ≠ _"*)
⟨*proof*⟩

theorem (**in** *lbvc*) *wtl_complete:*
assumes *wt: "wt_step r ⊤ step φ"*
and *s: "s <=_r φ!0" "s ∈ A" "s ≠ ⊤"*
and *len: "length ins = length phi"*
shows *"wtl ins c 0 s ≠ ⊤"*
⟨*proof*⟩

end

4.13 Abstract Bytecode Verifier

4.14 Semilattices

4.15 The Java Type System as Semilattice

theory *JType*
imports *"../DFA/Semilattices" "../J/WellForm"*
begin

definition *super :: "'a prog ⇒ cname ⇒ cname"* **where**
"super G C == fst (the (class G C))"

lemma *superI:*

```
  "G ⊢ C ≺C1 D ⟹ super G C = D"
```
⟨*proof*⟩

definition `is_ref :: "ty ⇒ bool"` **where**
```
  "is_ref T == case T of PrimT t ⇒ False | RefT r ⇒ True"
```

definition `sup :: "'c prog ⇒ ty ⇒ ty ⇒ ty err"` **where**
```
  "sup G T1 T2 ==
  case T1 of PrimT P1 ⇒ (case T2 of PrimT P2 ⇒
                          (if P1 = P2 then OK (PrimT P1) else Err) | RefT R ⇒ Err)
           | RefT R1 ⇒ (case T2 of PrimT P ⇒ Err | RefT R2 ⇒
  (case R1 of NullT ⇒ (case R2 of NullT ⇒ OK NT | ClassT C ⇒ OK (Class C))
            | ClassT C ⇒ (case R2 of NullT ⇒ OK (Class C)
                           | ClassT D ⇒ OK (Class (exec_lub (subcls1 G) (super G) C D)))))"
```

definition `subtype :: "'c prog ⇒ ty ⇒ ty ⇒ bool"` **where**
```
  "subtype G T1 T2 == G ⊢ T1 ≼ T2"
```

definition `is_ty :: "'c prog ⇒ ty ⇒ bool"` **where**
```
  "is_ty G T == case T of PrimT P ⇒ True | RefT R ⇒
                (case R of NullT ⇒ True | ClassT C ⇒ (C, Object) ∈ (subcls1 G)*)"
```

abbreviation `"types G == Collect (is_type G)"`

definition `esl :: "'c prog ⇒ ty esl"` **where**
```
  "esl G == (types G, subtype G, sup G)"
```

lemma `PrimT_PrimT: "(G ⊢ xb ≼ PrimT p) = (xb = PrimT p)"`
⟨*proof*⟩

lemma `PrimT_PrimT2: "(G ⊢ PrimT p ≼ xb) = (xb = PrimT p)"`
⟨*proof*⟩

lemma `is_tyI:`
```
  "⟦ is_type G T; ws_prog G ⟧ ⟹ is_ty G T"
```
⟨*proof*⟩

lemma `is_type_conv:`
```
  "ws_prog G ⟹ is_type G T = is_ty G T"
```
⟨*proof*⟩

lemma `order_widen:`
```
  "acyclic (subcls1 G) ⟹ order (subtype G)"
```
⟨*proof*⟩

lemma `wf_converse_subcls1_impl_acc_subtype:`
```
  "wf ((subcls1 G)^-1) ⟹ acc (subtype G)"
```
⟨*proof*⟩

lemma `closed_err_types:`
```
  "⟦ ws_prog G; single_valued (subcls1 G); acyclic (subcls1 G) ⟧
  ⟹ closed (err (types G)) (lift2 (sup G))"
```
⟨*proof*⟩

lemma *sup_subtype_greater:*
"⟦ *ws_prog G; single_valued (subcls1 G); acyclic (subcls1 G);*
is_type G t1; is_type G t2; sup G t1 t2 = OK s ⟧
⟹ *subtype G t1 s* ∧ *subtype G t2 s*"
⟨*proof*⟩

lemma *sup_subtype_smallest:*
"⟦ *ws_prog G; single_valued (subcls1 G); acyclic (subcls1 G);*
is_type G a; is_type G b; is_type G c;
subtype G a c; subtype G b c; sup G a b = OK d ⟧
⟹ *subtype G d c*"
⟨*proof*⟩

lemma *sup_exists:*
"⟦ *subtype G a c; subtype G b c; sup G a b = Err* ⟧ ⟹ *False*"
⟨*proof*⟩

lemma *err_semilat_JType_esl_lemma:*
"⟦ *ws_prog G; single_valued (subcls1 G); acyclic (subcls1 G)* ⟧
⟹ *err_semilat (esl G)*"
⟨*proof*⟩

lemma *single_valued_subcls1:*
"*ws_prog G* ⟹ *single_valued (subcls1 G)*"
⟨*proof*⟩

theorem *err_semilat_JType_esl:*
"*ws_prog G* ⟹ *err_semilat (esl G)*"
⟨*proof*⟩

end

4.16 The JVM Type System as Semilattice

theory *JVMType*
imports *JType*
begin

type_synonym *locvars_type = "ty err list"*
type_synonym *opstack_type = "ty list"*
type_synonym *state_type = "opstack_type* × *locvars_type"*
type_synonym *state = "state_type option err"* — for Kildall
type_synonym *method_type = "state_type option list"* — for BVSpec
type_synonym *class_type = "sig* ⇒ *method_type"*
type_synonym *prog_type = "cname* ⇒ *class_type"*

definition *stk_esl :: "'c prog* ⇒ *nat* ⇒ *ty list esl"* **where**
"stk_esl S maxs == upto_esl maxs (JType.esl S)"

definition *reg_sl :: "'c prog* ⇒ *nat* ⇒ *ty err list sl"* **where**
"reg_sl S maxr == Listn.sl maxr (Err.sl (JType.esl S))"

definition *sl :: "'c prog ⇒ nat ⇒ nat ⇒ state sl"* **where**
"sl S maxs maxr ==
Err.sl(Opt.esl(Product.esl (stk_esl S maxs) (Err.esl(reg_sl S maxr))))"

definition *states :: "'c prog ⇒ nat ⇒ nat ⇒ state set"* **where**
"states S maxs maxr == fst(sl S maxs maxr)"

definition *le :: "'c prog ⇒ nat ⇒ nat ⇒ state ord"* **where**
"le S maxs maxr == fst(snd(sl S maxs maxr))"

definition *sup :: "'c prog ⇒ nat ⇒ nat ⇒ state binop"* **where**
"sup S maxs maxr == snd(snd(sl S maxs maxr))"

definition *sup_ty_opt :: "['code prog,ty err,ty err] ⇒ bool"*
(<_ ⊢ _ <=o _> [71,71] 70) **where**
"sup_ty_opt G == Err.le (subtype G)"

definition *sup_loc :: "['code prog,locvars_type,locvars_type] ⇒ bool"*
(<_ ⊢ _ <=l _> [71,71] 70) **where**
"sup_loc G == Listn.le (sup_ty_opt G)"

definition *sup_state :: "['code prog,state_type,state_type] ⇒ bool"*
(<_ ⊢ _ <=s _> [71,71] 70) **where**
"sup_state G == Product.le (Listn.le (subtype G)) (sup_loc G)"

definition *sup_state_opt :: "['code prog,state_type option,state_type option] ⇒ bool"*

(<_ ⊢ _ <='' _> [71,71] 70) **where**
"sup_state_opt G == Opt.le (sup_state G)"

lemma *JVM_states_unfold:*
"states S maxs maxr == err(opt((⋃{list n (types S) |n. n <= maxs}) ×
list maxr (err(types S))))"
⟨*proof*⟩

lemma *JVM_le_unfold:*
"le S m n ==
Err.le(Opt.le(Product.le(Listn.le(subtype S))(Listn.le(Err.le(subtype S)))))"
⟨*proof*⟩

lemma *JVM_le_convert:*
"le G m n (OK t1) (OK t2) = G ⊢ t1 <=' t2"
⟨*proof*⟩

lemma *JVM_le_Err_conv:*
"le G m n = Err.le (sup_state_opt G)"
⟨*proof*⟩

lemma *zip_map [rule_format]:*
"∀a. length a = length b ⟶
zip (map f a) (map g b) = map (λ(x,y). (f x, g y)) (zip a b)"

⟨*proof*⟩

lemma *[simp]: "Err.le r (OK a) (OK b) = r a b"*
⟨*proof*⟩

lemma *stk_convert:*
"Listn.le (subtype G) a b = G ⊢ map OK a <=l map OK b"
⟨*proof*⟩

lemma *sup_state_conv:*
"(G ⊢ s1 <=s s2) ==
(G ⊢ map OK (fst s1) <=l map OK (fst s2)) ∧ (G ⊢ snd s1 <=l snd s2)"
⟨*proof*⟩

lemma *subtype_refl [simp]:*
"subtype G t t"
⟨*proof*⟩

theorem *sup_ty_opt_refl [simp]:*
"G ⊢ t <=o t"
⟨*proof*⟩

lemma *le_list_refl2 [simp]:*
"(⋀xs. r xs xs) ⟹ Listn.le r xs xs"
⟨*proof*⟩

theorem *sup_loc_refl [simp]:*
"G ⊢ t <=l t"
⟨*proof*⟩

theorem *sup_state_refl [simp]:*
"G ⊢ s <=s s"
⟨*proof*⟩

theorem *sup_state_opt_refl [simp]:*
"G ⊢ s <=' s"
⟨*proof*⟩

theorem *anyConvErr [simp]:*
"(G ⊢ Err <=o any) = (any = Err)"
⟨*proof*⟩

theorem *OKanyConvOK [simp]:*
"(G ⊢ (OK ty') <=o (OK ty)) = (G ⊢ ty' ⪯ ty)"
⟨*proof*⟩

theorem *sup_ty_opt_OK:*
"G ⊢ a <=o (OK b) ⟹ ∃ x. a = OK x"
⟨*proof*⟩

lemma *widen_PrimT_conv1 [simp]:*

"⟦ G ⊢ S ≼ T; S = PrimT x⟧ ⟹ T = PrimT x"
⟨proof⟩

theorem *sup_PTS_eq:*
"(G ⊢ OK (PrimT p) <=o X) = (X=Err ∨ X = OK (PrimT p))"
⟨proof⟩

theorem *sup_loc_Nil [iff]:*
"(G ⊢ [] <=l XT) = (XT=[])"
⟨proof⟩

theorem *sup_loc_Cons [iff]:*
"(G ⊢ (Y#YT) <=l XT) = (∃X XT'. XT=X#XT' ∧ (G ⊢ Y <=o X) ∧ (G ⊢ YT <=l XT'))"
⟨proof⟩

theorem *sup_loc_Cons2:*
"(G ⊢ YT <=l (X#XT)) = (∃Y YT'. YT=Y#YT' ∧ (G ⊢ Y <=o X) ∧ (G ⊢ YT' <=l XT))"
⟨proof⟩

lemma *sup_state_Cons:*
"(G ⊢ (x#xt, a) <=s (y#yt, b)) =
((G ⊢ x ≼ y) ∧ (G ⊢ (xt,a) <=s (yt,b)))"
⟨proof⟩

theorem *sup_loc_length:*
"G ⊢ a <=l b ⟹ length a = length b"
⟨proof⟩

theorem *sup_loc_nth:*
"⟦ G ⊢ a <=l b; n < length a ⟧ ⟹ G ⊢ (a!n) <=o (b!n)"
⟨proof⟩

theorem *all_nth_sup_loc:*
"∀b. length a = length b ⟶ (∀ n. n < length a ⟶ (G ⊢ (a!n) <=o (b!n)))
⟶ (G ⊢ a <=l b)" (**is** "?P a")
⟨proof⟩

theorem *sup_loc_append:*
"length a = length b ⟹
(G ⊢ (a@x) <=l (b@y)) = ((G ⊢ a <=l b) ∧ (G ⊢ x <=l y))"
⟨proof⟩

theorem *sup_loc_rev [simp]:*
"(G ⊢ (rev a) <=l rev b) = (G ⊢ a <=l b)"
⟨proof⟩

theorem *sup_loc_update [rule_format]:*
"∀ n y. (G ⊢ a <=o b) ⟶ n < length y ⟶ (G ⊢ x <=l y) ⟶
(G ⊢ x[n := a] <=l y[n := b])" (**is** "?P x")
⟨proof⟩

theorem *sup_state_length [simp]:*
"G ⊢ s2 <=s s1 ⟹
length (fst s2) = length (fst s1) ∧ length (snd s2) = length (snd s1)"
⟨*proof*⟩

theorem *sup_state_append_snd:*
"length a = length b ⟹
(G ⊢ (i,a@x) <=s (j,b@y)) = ((G ⊢ (i,a) <=s (j,b)) ∧ (G ⊢ (i,x) <=s (j,y)))"
⟨*proof*⟩

theorem *sup_state_append_fst:*
"length a = length b ⟹
(G ⊢ (a@x,i) <=s (b@y,j)) = ((G ⊢ (a,i) <=s (b,j)) ∧ (G ⊢ (x,i) <=s (y,j)))"
⟨*proof*⟩

theorem *sup_state_Cons1:*
"(G ⊢ (x#xt, a) <=s (yt, b)) =
(∃y yt'. yt=y#yt' ∧ (G ⊢ x ≼ y) ∧ (G ⊢ (xt,a) <=s (yt',b)))"
⟨*proof*⟩

theorem *sup_state_Cons2:*
"(G ⊢ (xt, a) <=s (y#yt, b)) =
(∃x xt'. xt=x#xt' ∧ (G ⊢ x ≼ y) ∧ (G ⊢ (xt',a) <=s (yt,b)))"
⟨*proof*⟩

theorem *sup_state_ignore_fst:*
"G ⊢ (a, x) <=s (b, y) ⟹ G ⊢ (c, x) <=s (c, y)"
⟨*proof*⟩

theorem *sup_state_rev_fst:*
"(G ⊢ (rev a, x) <=s (rev b, y)) = (G ⊢ (a, x) <=s (b, y))"
⟨*proof*⟩

lemma *sup_state_opt_None_any [iff]:*
"(G ⊢ None <=' any) = True"
⟨*proof*⟩

lemma *sup_state_opt_any_None [iff]:*
"(G ⊢ any <=' None) = (any = None)"
⟨*proof*⟩

lemma *sup_state_opt_Some_Some [iff]:*
"(G ⊢ (Some a) <=' (Some b)) = (G ⊢ a <=s b)"
⟨*proof*⟩

lemma *sup_state_opt_any_Some [iff]:*
"(G ⊢ (Some a) <=' any) = (∃b. any = Some b ∧ G ⊢ a <=s b)"
⟨*proof*⟩

lemma *sup_state_opt_Some_any:*
"(G ⊢ any <=' (Some b)) = (any = None ∨ (∃a. any = Some a ∧ G ⊢ a <=s b))"
⟨*proof*⟩

theorem *sup_ty_opt_trans [trans]:*
"⟦G ⊢ a <=o b; G ⊢ b <=o c⟧ ⟹ G ⊢ a <=o c"
⟨*proof*⟩

theorem *sup_loc_trans [trans]:*
"⟦G ⊢ a <=l b; G ⊢ b <=l c⟧ ⟹ G ⊢ a <=l c"
⟨*proof*⟩

theorem *sup_state_trans [trans]:*
"⟦G ⊢ a <=s b; G ⊢ b <=s c⟧ ⟹ G ⊢ a <=s c"
⟨*proof*⟩

theorem *sup_state_opt_trans [trans]:*
"⟦G ⊢ a <=' b; G ⊢ b <=' c⟧ ⟹ G ⊢ a <=' c"
⟨*proof*⟩

end

4.17 Effect of Instructions on the State Type

theory *Effect*
imports *JVMType "../JVM/JVMExceptions"*
begin

type_synonym *succ_type = "(p_count × state_type option) list"*

Program counter of successor instructions:

primrec *succs :: "instr ⇒ p_count ⇒ p_count list"* **where**
"succs (Load idx) pc = [pc+1]"
| "succs (Store idx) pc = [pc+1]"
| "succs (LitPush v) pc = [pc+1]"
| "succs (Getfield F C) pc = [pc+1]"
| "succs (Putfield F C) pc = [pc+1]"
| "succs (New C) pc = [pc+1]"
| "succs (Checkcast C) pc = [pc+1]"
| "succs Pop pc = [pc+1]"
| "succs Dup pc = [pc+1]"
| "succs Dup_x1 pc = [pc+1]"
| "succs Dup_x2 pc = [pc+1]"
| "succs Swap pc = [pc+1]"
| "succs IAdd pc = [pc+1]"
| "succs (Ifcmpeq b) pc = [pc+1, nat (int pc + b)]"
| "succs (Goto b) pc = [nat (int pc + b)]"
| "succs Return pc = [pc]"
| "succs (Invoke C mn fpTs) pc = [pc+1]"
| "succs Throw pc = [pc]"

Effect of instruction on the state type:

fun *eff' :: "instr × jvm_prog × state_type ⇒ state_type"*
where

```
"eff' (Load idx,  G, (ST, LT))          = (ok_val (LT ! idx) # ST, LT)" |
"eff' (Store idx, G, (ts#ST, LT))       = (ST, LT[idx:= OK ts])" |
"eff' (LitPush v, G, (ST, LT))           = (the (typeof (λv. None) v) # ST, LT)" |
"eff' (Getfield F C, G, (oT#ST, LT))    = (snd (the (field (G,C) F)) # ST, LT)" |
"eff' (Putfield F C, G, (vT#oT#ST, LT)) = (ST,LT)" |
"eff' (New C, G, (ST,LT))               = (Class C # ST, LT)" |
"eff' (Checkcast C, G, (RefT rt#ST,LT)) = (Class C # ST,LT)" |
"eff' (Pop, G, (ts#ST,LT))              = (ST,LT)" |
"eff' (Dup, G, (ts#ST,LT))              = (ts#ts#ST,LT)" |
"eff' (Dup_x1, G, (ts1#ts2#ST,LT))      = (ts1#ts2#ts1#ST,LT)" |
"eff' (Dup_x2, G, (ts1#ts2#ts3#ST,LT))  = (ts1#ts2#ts3#ts1#ST,LT)" |
"eff' (Swap, G, (ts1#ts2#ST,LT))        = (ts2#ts1#ST,LT)" |
"eff' (IAdd, G, (PrimT Integer#PrimT Integer#ST,LT))
                                         = (PrimT Integer#ST,LT)" |
"eff' (Ifcmpeq b, G, (ts1#ts2#ST,LT))   = (ST,LT)" |
"eff' (Goto b, G, s)                    = s" |
```

— Return has no successor instruction in the same method

```
"eff' (Return, G, s)                    = s" |
```

— Throw always terminates abruptly

```
"eff' (Throw, G, s)                     = s" |
"eff' (Invoke C mn fpTs, G, (ST,LT))    = (let ST' = drop (length fpTs) ST
  in  (fst (snd (the (method (G,C) (mn,fpTs))))#(tl ST'),LT))"
```

primrec *match_any :: "jvm_prog ⇒ p_count ⇒ exception_table ⇒ cname list"* **where**

```
  "match_any G pc [] = []"
| "match_any G pc (e#es) = (let (start_pc, end_pc, handler_pc, catch_type) = e;
                                es' = match_any G pc es
                            in
                            if start_pc <= pc ∧ pc < end_pc then catch_type#es' else es')"
```

primrec *match :: "jvm_prog ⇒ xcpt ⇒ p_count ⇒ exception_table ⇒ cname list"* **where**

```
  "match G X pc [] = []"
| "match G X pc (e#es) =
  (if match_exception_entry G (Xcpt X) pc e then [Xcpt X] else match G X pc es)"
```

lemma *match_some_entry:*

```
  "match G X pc et = (if ∃e ∈ set et. match_exception_entry G (Xcpt X) pc e then [Xcpt
X] else [])"
```

⟨*proof*⟩

fun

```
  xcpt_names :: "instr × jvm_prog × p_count × exception_table ⇒ cname list"
```

where

```
  "xcpt_names (Getfield F C, G, pc, et) = match G NullPointer pc et"
| "xcpt_names (Putfield F C, G, pc, et) = match G NullPointer pc et"
| "xcpt_names (New C, G, pc, et)        = match G OutOfMemory pc et"
| "xcpt_names (Checkcast C, G, pc, et)  = match G ClassCast pc et"
| "xcpt_names (Throw, G, pc, et)        = match_any G pc et"
| "xcpt_names (Invoke C m p, G, pc, et) = match_any G pc et"
| "xcpt_names (i, G, pc, et)            = []"
```

definition *xcpt_eff :: "instr ⇒ jvm_prog ⇒ p_count ⇒ state_type option ⇒ exception_table ⇒ succ_type"* **where**
"xcpt_eff i G pc s et ==
map (λC. (the (match_exception_table G C pc et), case s of None ⇒ None | Some s' ⇒ Some ([Class C], snd s')))
(xcpt_names (i,G,pc,et))"

definition *norm_eff :: "instr ⇒ jvm_prog ⇒ state_type option ⇒ state_type option"* **where**
"norm_eff i G == map_option (λs. eff' (i,G,s))"

definition *eff :: "instr ⇒ jvm_prog ⇒ p_count ⇒ exception_table ⇒ state_type option ⇒ succ_type"* **where**
"eff i G pc et s == (map (λpc'. (pc',norm_eff i G s)) (succs i pc)) @ (xcpt_eff i G pc s et)"

definition *isPrimT :: "ty ⇒ bool"* **where**
"isPrimT T == case T of PrimT T' ⇒ True | RefT T' ⇒ False"

definition *isRefT :: "ty ⇒ bool"* **where**
"isRefT T == case T of PrimT T' ⇒ False | RefT T' ⇒ True"

lemma *isPrimT [simp]:*
"isPrimT T = (∃T'. T = PrimT T')" ⟨*proof*⟩

lemma *isRefT [simp]:*
"isRefT T = (∃T'. T = RefT T')" ⟨*proof*⟩

lemma *"list_all2 P a b ⟹ ∀(x,y) ∈ set (zip a b). P x y"*
⟨*proof*⟩

Conditions under which eff is applicable:

fun
app' :: "instr × jvm_prog × p_count × nat × ty × state_type ⇒ bool"
where
"app' (Load idx, G, pc, maxs, rT, s) =
(idx < length (snd s) ∧ (snd s) ! idx ≠ Err ∧ length (fst s) < maxs)" |
"app' (Store idx, G, pc, maxs, rT, (ts#ST, LT)) =
(idx < length LT)" |
"app' (LitPush v, G, pc, maxs, rT, s) =
(length (fst s) < maxs ∧ typeof (λt. None) v ≠ None)" |
"app' (Getfield F C, G, pc, maxs, rT, (oT#ST, LT)) =
(is_class G C ∧ field (G,C) F ≠ None ∧ fst (the (field (G,C) F)) = C ∧
G ⊢ oT ⪯ (Class C))" |
"app' (Putfield F C, G, pc, maxs, rT, (vT#oT#ST, LT)) =
(is_class G C ∧ field (G,C) F ≠ None ∧ fst (the (field (G,C) F)) = C ∧
G ⊢ oT ⪯ (Class C) ∧ G ⊢ vT ⪯ (snd (the (field (G,C) F))))" |
"app' (New C, G, pc, maxs, rT, s) =
(is_class G C ∧ length (fst s) < maxs)" |
"app' (Checkcast C, G, pc, maxs, rT, (RefT rt#ST,LT)) =
(is_class G C)" |
"app' (Pop, G, pc, maxs, rT, (ts#ST,LT)) =
True" |
"app' (Dup, G, pc, maxs, rT, (ts#ST,LT)) =

```
  (1+length ST < maxs)" |
"app' (Dup_x1, G, pc, maxs, rT, (ts1#ts2#ST,LT)) =
  (2+length ST < maxs)" |
"app' (Dup_x2, G, pc, maxs, rT, (ts1#ts2#ts3#ST,LT)) =
  (3+length ST < maxs)" |
"app' (Swap, G, pc, maxs, rT, (ts1#ts2#ST,LT)) =
  True" |
"app' (IAdd, G, pc, maxs, rT, (PrimT Integer#PrimT Integer#ST,LT)) =
  True" |
"app' (Ifcmpeq b, G, pc, maxs, rT, (ts#ts'#ST,LT)) =
  (0 ≤ int pc + b ∧ (isPrimT ts ∧ ts' = ts ∨ isRefT ts ∧ isRefT ts'))" |
"app' (Goto b, G, pc, maxs, rT, s) =
  (0 ≤ int pc + b)" |
"app' (Return, G, pc, maxs, rT, (T#ST,LT)) =
  (G ⊢ T ≼ rT)" |
"app' (Throw, G, pc, maxs, rT, (T#ST,LT)) =
  isRefT T" |
"app' (Invoke C mn fpTs, G, pc, maxs, rT, s) =
  (length fpTs < length (fst s) ∧
  (let apTs = rev (take (length fpTs) (fst s));
       X    = hd (drop (length fpTs) (fst s))
   in
       G ⊢ X ≼ Class C ∧ is_class G C ∧ method (G,C) (mn,fpTs) ≠ None ∧
       list_all2 (λx y. G ⊢ x ≼ y) apTs fpTs))" |

"app' (i,G, pc,maxs,rT,s) = False"
```

definition *xcpt_app :: "instr ⇒ jvm_prog ⇒ nat ⇒ exception_table ⇒ bool"* **where**
"xcpt_app i G pc et ≡ ∀C∈set(xcpt_names (i,G,pc,et)). is_class G C"

definition *app :: "instr ⇒ jvm_prog ⇒ nat ⇒ ty ⇒ nat ⇒ exception_table ⇒ state_type option ⇒ bool"* **where**
"app i G maxs rT pc et s == case s of None ⇒ True | Some t ⇒ app' (i,G,pc,maxs,rT,t) ∧ xcpt_app i G pc et"

lemma *match_any_match_table:*
"C ∈ set (match_any G pc et) ⟹ match_exception_table G C pc et ≠ None"
⟨*proof*⟩

lemma *match_X_match_table:*
"C ∈ set (match G X pc et) ⟹ match_exception_table G C pc et ≠ None"
⟨*proof*⟩

lemma *xcpt_names_in_et:*
"C ∈ set (xcpt_names (i,G,pc,et)) ⟹
∃e ∈ set et. the (match_exception_table G C pc et) = fst (snd (snd e))"
⟨*proof*⟩

lemma *1: "2 < length a ⟹ (∃l l' l'' ls. a = l#l'#l''#ls)"*
⟨*proof*⟩

lemma *2: "¬(2 < length a) ⟹ a = [] ∨ (∃ l. a = [l]) ∨ (∃ l l'. a = [l,l'])"*

⟨*proof*⟩

lemmas *[simp] = app_def xcpt_app_def*

simp rules for *app*

lemma *appNone[simp]: "app i G maxs rT pc et None = True"* ⟨*proof*⟩

lemma *appLoad[simp]:*
"(app (Load idx) G maxs rT pc et (Some s)) = (∃ST LT. s = (ST,LT) ∧ idx < length LT ∧ LT!idx ≠ Err ∧ length ST < maxs)"
⟨*proof*⟩

lemma *appStore[simp]:*
"(app (Store idx) G maxs rT pc et (Some s)) = (∃ts ST LT. s = (ts#ST,LT) ∧ idx < length LT)"
⟨*proof*⟩

lemma *appLitPush[simp]:*
"(app (LitPush v) G maxs rT pc et (Some s)) = (∃ST LT. s = (ST,LT) ∧ length ST < maxs ∧ typeof (λv. None) v ≠ None)"
⟨*proof*⟩

lemma *appGetField[simp]:*
"(app (Getfield F C) G maxs rT pc et (Some s)) =
(∃ oT vT ST LT. s = (oT#ST, LT) ∧ is_class G C ∧
field (G,C) F = Some (C,vT) ∧ G ⊢ oT ⪯ (Class C) ∧ (∀x ∈ set (match G NullPointer pc et). is_class G x))"
⟨*proof*⟩

lemma *appPutField[simp]:*
"(app (Putfield F C) G maxs rT pc et (Some s)) =
(∃ vT vT' oT ST LT. s = (vT#oT#ST, LT) ∧ is_class G C ∧
field (G,C) F = Some (C, vT') ∧ G ⊢ oT ⪯ (Class C) ∧ G ⊢ vT ⪯ vT' ∧
(∀x ∈ set (match G NullPointer pc et). is_class G x))"
⟨*proof*⟩

lemma *appNew[simp]:*
"(app (New C) G maxs rT pc et (Some s)) =
(∃ST LT. s=(ST,LT) ∧ is_class G C ∧ length ST < maxs ∧
(∀x ∈ set (match G OutOfMemory pc et). is_class G x))"
⟨*proof*⟩

lemma *appCheckcast[simp]:*
"(app (Checkcast C) G maxs rT pc et (Some s)) =
(∃rT ST LT. s = (RefT rT#ST,LT) ∧ is_class G C ∧
(∀x ∈ set (match G ClassCast pc et). is_class G x))"
⟨*proof*⟩

lemma *appPop[simp]:*
"(app Pop G maxs rT pc et (Some s)) = (∃ts ST LT. s = (ts#ST,LT))"
⟨*proof*⟩

lemma *appDup[simp]:*
```
"(app Dup G maxs rT pc et (Some s)) = (∃ts ST LT. s = (ts#ST,LT) ∧ 1+length ST < maxs)"
```

⟨*proof*⟩

lemma *appDup_x1[simp]:*
```
"(app Dup_x1 G maxs rT pc et (Some s)) = (∃ts1 ts2 ST LT. s = (ts1#ts2#ST,LT) ∧ 2+length
ST < maxs)"
```
⟨*proof*⟩

lemma *appDup_x2[simp]:*
```
"(app Dup_x2 G maxs rT pc et (Some s)) = (∃ts1 ts2 ts3 ST LT. s = (ts1#ts2#ts3#ST,LT)
∧ 3+length ST < maxs)"
```
⟨*proof*⟩

lemma *appSwap[simp]:*
```
"app Swap G maxs rT pc et (Some s) = (∃ts1 ts2 ST LT. s = (ts1#ts2#ST,LT))"
```
⟨*proof*⟩

lemma *appIAdd[simp]:*
```
"app IAdd G maxs rT pc et (Some s) = (∃ ST LT. s = (PrimT Integer#PrimT Integer#ST,LT))"
  (is "?app s = ?P s")
```
⟨*proof*⟩

lemma *appIfcmpeq[simp]:*
```
"app (Ifcmpeq b) G maxs rT pc et (Some s) =
  (∃ts1 ts2 ST LT. s = (ts1#ts2#ST,LT) ∧ 0 ≤ int pc + b ∧
  ((∃ p. ts1 = PrimT p ∧ ts2 = PrimT p) ∨ (∃r r'. ts1 = RefT r ∧ ts2 = RefT r')))"
```
⟨*proof*⟩

lemma *appReturn[simp]:*
```
"app Return G maxs rT pc et (Some s) = (∃T ST LT. s = (T#ST,LT) ∧ (G ⊢ T ≼ rT))"
```
⟨*proof*⟩

lemma *appGoto[simp]:*
```
"app (Goto b) G maxs rT pc et (Some s) = (0 ≤ int pc + b)"
```
⟨*proof*⟩

lemma *appThrow[simp]:*
```
  "app Throw G maxs rT pc et (Some s) =
  (∃T ST LT r. s=(T#ST,LT) ∧ T = RefT r ∧ (∀C ∈ set (match_any G pc et). is_class G
C))"
```
⟨*proof*⟩

lemma *appInvoke[simp]:*
```
"app (Invoke C mn fpTs) G maxs rT pc et (Some s) = (∃apTs X ST LT mD' rT' b'.
  s = ((rev apTs) @ (X # ST), LT) ∧ length apTs = length fpTs ∧ is_class G C ∧
  G ⊢ X ≼ Class C ∧ (∀(aT,fT)∈set(zip apTs fpTs). G ⊢ aT ≼ fT) ∧
  method (G,C) (mn,fpTs) = Some (mD', rT', b') ∧
```

```
  (∀C ∈ set (match_any G pc et). is_class G C))" (is "?app s = ?P s")
```
⟨*proof*⟩

lemma *effNone:*
```
  "(pc', s') ∈ set (eff i G pc et None) ⟹ s' = None"
```
⟨*proof*⟩

lemma *xcpt_app_lemma [code]:*
```
  "xcpt_app i G pc et = list_all (is_class G) (xcpt_names (i, G, pc, et))"
```
⟨*proof*⟩

lemmas *[simp del] = app_def xcpt_app_def*

end

4.18 Monotonicity of eff and app

theory *EffectMono*
imports *Effect*
begin

lemma *PrimT_PrimT: "(G ⊢ xb ⪯ PrimT p) = (xb = PrimT p)"*
⟨*proof*⟩

lemma *sup_loc_some [rule_format]:*
```
"∀y n. (G ⊢ b <=l y) ⟶ n < length y ⟶ y!n = OK t ⟶
  (∃t. b!n = OK t ∧ (G ⊢ (b!n) <=o (y!n)))"
```
⟨*proof*⟩

lemma *all_widen_is_sup_loc:*
```
"∀b. length a = length b ⟶
     (∀(x, y)∈set (zip a b). G ⊢ x ⪯ y) = (G ⊢ (map OK a) <=l (map OK b))"
 (is "∀b. length a = length b ⟶ ?Q a b" is "?P a")
```
⟨*proof*⟩

lemma *append_length_n [rule_format]:*
```
"∀n. n ≤ length x ⟶ (∃a b. x = a@b ∧ length a = n)"
```
⟨*proof*⟩

lemma *rev_append_cons:*
```
"n < length x ⟹ ∃a b c. x = (rev a) @ b # c ∧ length a = n"
```
⟨*proof*⟩

lemma *sup_loc_length_map:*
```
  "G ⊢ map f a <=l map g b ⟹ length a = length b"
```
⟨*proof*⟩

lemmas *[iff] = not_Err_eq*

lemma *app_mono:*
"⟦G ⊢ s <=' s'; app i G m rT pc et s'⟧ ⟹ app i G m rT pc et s"
⟨*proof*⟩

lemmas *[simp del] = split_paired_Ex*

lemma *eff'_mono:*
"⟦ app i G m rT pc et (Some s2); G ⊢ s1 <=s s2 ⟧ ⟹
  G ⊢ eff' (i,G,s1) <=s eff' (i,G,s2)"
⟨*proof*⟩

lemmas *[iff del] = not_Err_eq*

end

4.19 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 program counter will always be inside the method:
check_bounded :: "instr list ⇒ exception_table ⇒ bool" **where**
"check_bounded ins et ⟷
(∀pc < length ins. ∀pc' ∈ set (succs (ins!pc) pc). pc' < length ins) ∧
(∀e ∈ set et. fst (snd (snd e)) < length ins)"

definition
— The method type only contains declared classes:
check_types :: "jvm_prog ⇒ nat ⇒ nat ⇒ JVMType.state list ⇒ bool" **where**
"check_types G mxs mxr phi ⟷ set phi ⊆ states G mxs mxr"

definition
— An instruction is welltyped if it is applicable and its effect
— is compatible with the type at all successor instructions:
wt_instr :: "[instr,jvm_prog,ty,method_type,nat,p_count,
exception_table,p_count] ⇒ bool" **where**
"wt_instr i G rT phi mxs max_pc et pc ⟷
app i G mxs rT pc et (phi!pc) ∧
(∀(pc',s') ∈ set (eff i G pc et (phi!pc)). pc' < max_pc ∧ G ⊢ s' <=' phi!pc')"

definition
— The type at *pc=0* conforms to the method calling convention:
wt_start :: "[jvm_prog,cname,ty list,nat,method_type] ⇒ bool" **where**
"wt_start G C pTs mxl phi ⟷
G ⊢ Some ([],(OK (Class C))#((map OK pTs))@(replicate mxl Err)) <=' phi!0"

definition
— A method is welltyped if the body is not empty, if execution does not
— leave the body, if the method type covers all instructions and mentions

— declared classes only, if the method calling convention is respected, and
— if all instructions are welltyped.

```
wt_method :: "[jvm_prog,cname,ty list,ty,nat,nat,instr list,
               exception_table,method_type] ⇒ bool" where
"wt_method G C pTs rT mxs mxl ins et phi ⟷
(let max_pc = length ins in
0 < max_pc ∧
length phi = length ins ∧
check_bounded ins et ∧
check_types G mxs (1+length pTs+mxl) (map OK phi) ∧
wt_start G C pTs mxl phi ∧
(∀pc. pc<max_pc ⟶ wt_instr (ins!pc) G rT phi mxs max_pc et pc))"
```

definition

— A program is welltyped if it is wellformed and all methods are welltyped

```
wt_jvm_prog :: "[jvm_prog,prog_type] ⇒ bool" where
"wt_jvm_prog G phi ⟷
wf_prog (λG C (sig,rT,(maxs,maxl,b,et)).
         wt_method G C (snd sig) rT maxs maxl b et (phi C sig)) G"
```

lemma *check_boundedD:*

```
"⟦ check_bounded ins et; pc < length ins;
  (pc',s') ∈ set (eff (ins!pc) G pc et s)  ⟧ ⟹
pc' < length ins"
```

⟨*proof*⟩

lemma *wt_jvm_progD:*

```
"wt_jvm_prog G phi ⟹ (∃wt. wf_prog wt G)"
```

⟨*proof*⟩

lemma *wt_jvm_prog_impl_wt_instr:*

```
"⟦ wt_jvm_prog G phi; is_class G C;
    method (G,C) sig = Some (C,rT,maxs,maxl,ins,et); pc < length ins ⟧
⟹ wt_instr (ins!pc) G rT (phi C sig) maxs (length ins) et pc"
```

⟨*proof*⟩

We could leave out the check `pc' < max_pc` in the definition of `wt_instr` in the context of `wt_method`.

lemma *wt_instr_def2:*

```
"⟦ wt_jvm_prog G Phi; is_class G C;
    method (G,C) sig = Some (C,rT,maxs,maxl,ins,et); pc < length ins;
    i = ins!pc; phi = Phi C sig; max_pc = length ins ⟧
⟹ wt_instr i G rT phi maxs max_pc et pc =
   (app i G maxs rT pc et (phi!pc) ∧
   (∀(pc',s') ∈ set (eff i G pc et (phi!pc)). G ⊢ s' <=' phi!pc'))"
```

⟨*proof*⟩

lemma *wt_jvm_prog_impl_wt_start:*

```
"⟦ wt_jvm_prog G phi; is_class G C;
    method (G,C) sig = Some (C,rT,maxs,maxl,ins,et) ⟧ ⟹
0 < (length ins) ∧ wt_start G C (snd sig) maxl (phi C sig)"
```

⟨*proof*⟩

end

4.20 The Typing Framework for the JVM

theory *Typing_Framework_JVM*
imports *"../DFA/Abstract_BV" JVMType EffectMono BVSpec*
begin

definition *exec :: "jvm_prog ⇒ nat ⇒ ty ⇒ exception_table ⇒ instr list ⇒ JVMType.state step_type"* **where**
"exec G maxs rT et bs ==
err_step (size bs) (λpc. app (bs!pc) G maxs rT pc et) (λpc. eff (bs!pc) G pc et)"

definition *opt_states :: "'c prog ⇒ nat ⇒ nat ⇒ (ty list × ty err list) option set"* **where**
"opt_states G maxs maxr ≡ opt (⋃{list n (types G) |n. n ≤ maxs} × list maxr (err (types G)))"

4.20.1 Executability of *check_bounded*

primrec *list_all'_rec :: "('a ⇒ nat ⇒ bool) ⇒ nat ⇒ 'a list ⇒ bool"*
where
"list_all'_rec P n []   = True"
| "list_all'_rec P n (x#xs) = (P x n ∧ list_all'_rec P (Suc n) xs)"

definition *list_all' :: "('a ⇒ nat ⇒ bool) ⇒ 'a list ⇒ bool"* **where**
"list_all' P xs ≡ list_all'_rec P 0 xs"

lemma *list_all'_rec:*
"list_all'_rec P n xs = (∀p < size xs. P (xs!p) (p+n))"
⟨*proof*⟩

lemma *list_all' [iff]:*
"list_all' P xs = (∀n < size xs. P (xs!n) n)"
⟨*proof*⟩

lemma *[code]:*
"check_bounded ins et =
(list_all' (λi pc. list_all (λpc'. pc' < length ins) (succs i pc)) ins ∧
list_all (λe. fst (snd (snd e)) < length ins) et)"
⟨*proof*⟩

4.20.2 Connecting JVM and Framework

lemma *check_bounded_is_bounded:*
"check_bounded ins et ⟹ bounded (λpc. eff (ins!pc) G pc et) (length ins)"
⟨*proof*⟩

lemma *special_ex_swap_lemma [iff]:*
"(∃X. (∃n. X = A n ∧ P n) & Q X) = (∃n. Q(A n) ∧ P n)"
⟨*proof*⟩

lemmas *[iff del] = not_None_eq*

theorem *exec_pres_type:*
"wf_prog wf_mb S ⟹
pres_type (exec S maxs rT et bs) (size bs) (states S maxs maxr)"
⟨*proof*⟩

lemmas *[iff] = not_None_eq*

lemma *sup_state_opt_unfold:*
"sup_state_opt G ≡ Opt.le (Product.le (Listn.le (subtype G)) (Listn.le (Err.le (subtype G))))"
⟨*proof*⟩

lemma *app_mono:*
"app_mono (sup_state_opt G) (λpc. app (bs!pc) G maxs rT pc et) (length bs) (opt_states G maxs maxr)"
⟨*proof*⟩

lemma *list_appendI:*
"⟦a ∈ list x A; b ∈ list y A⟧ ⟹ a @ b ∈ list (x+y) A"
⟨*proof*⟩

lemma *list_map [simp]:*
"(map f xs ∈ list (length xs) A) = (f ‘ set xs ⊆ A)"
⟨*proof*⟩

lemma *[iff]:*
"(OK ‘ A ⊆ err B) = (A ⊆ B)"
⟨*proof*⟩

lemma *[intro]:*
"x ∈ A ⟹ replicate n x ∈ list n A"
⟨*proof*⟩

lemma *lesubstep_type_simple:*
"a <=[Product.le (=) r] b ⟹ a ≤|r| b"
⟨*proof*⟩

lemma *eff_mono:*
"⟦p < length bs; s <=_(sup_state_opt G) t; app (bs!p) G maxs rT pc et t⟧
⟹ eff (bs!p) G p et s ≤|sup_state_opt G| eff (bs!p) G p et t"
⟨*proof*⟩

lemma *order_sup_state_opt:*
"ws_prog G ⟹ order (sup_state_opt G)"
⟨*proof*⟩

theorem *exec_mono:*
"ws_prog G ⟹ bounded (exec G maxs rT et bs) (size bs) ⟹
mono (JVMType.le G maxs maxr) (exec G maxs rT et bs) (size bs) (states G maxs maxr)"

⟨*proof*⟩

theorem *semilat_JVM_slI:*
"ws_prog G ⟹ semilat (JVMType.sl G maxs maxr)"
⟨*proof*⟩

lemma *sl_triple_conv:*
"JVMType.sl G maxs maxr ==
(states G maxs maxr, JVMType.le G maxs maxr, JVMType.sup G maxs maxr)"
⟨*proof*⟩

lemma *is_type_pTs:*
"⟦ wf_prog wf_mb G; (C,S,fs,mdecls) ∈ set G; ((mn,pTs),rT,code) ∈ set mdecls ⟧
⟹ set pTs ⊆ types G"
⟨*proof*⟩

lemma *jvm_prog_lift:*
assumes *wf:*
"wf_prog (λG C bd. P G C bd) G"

assumes *rule:*
"⋀wf_mb C mn pTs C rT maxs maxl b et bd.
wf_prog wf_mb G ⟹
method (G,C) (mn,pTs) = Some (C,rT,maxs,maxl,b,et) ⟹
is_class G C ⟹
set pTs ⊆ types G ⟹
bd = ((mn,pTs),rT,maxs,maxl,b,et) ⟹
P G C bd ⟹
Q G C bd"

shows
"wf_prog (λG C bd. Q G C bd) G"
⟨*proof*⟩

end

4.21 LBV for the JVM

theory *LBVJVM*
imports *Typing_Framework_JVM*
begin

type_synonym *prog_cert = "cname ⇒ sig ⇒ JVMType.state list"*

definition *check_cert :: "jvm_prog ⇒ nat ⇒ nat ⇒ nat ⇒ JVMType.state list ⇒ bool"*
where
"check_cert G mxs mxr n cert ≡ check_types G mxs mxr cert ∧ length cert = n+1 ∧
(∀ i<n. cert!i ≠ Err) ∧ cert!n = OK None"

definition *lbvjvm :: "jvm_prog ⇒ nat ⇒ nat ⇒ ty ⇒ exception_table ⇒*
JVMType.state list ⇒ instr list ⇒ JVMType.state ⇒ JVMType.state" **where**
"lbvjvm G maxs maxr rT et cert bs ≡
wtl_inst_list bs cert (JVMType.sup G maxs maxr) (JVMType.le G maxs maxr) Err (OK None)

```
(exec G maxs rT et bs) 0"

definition wt_lbv :: "jvm_prog ⇒ cname ⇒ ty list ⇒ ty ⇒ nat ⇒ nat ⇒
            exception_table ⇒ JVMType.state list ⇒ instr list ⇒ bool" where
  "wt_lbv G C pTs rT mxs mxl et cert ins ≡
   check_bounded ins et ∧
   check_cert G mxs (1+size pTs+mxl) (length ins) cert ∧
   0 < size ins ∧
   (let start  = Some ([],(OK (Class C))#((map OK pTs))@(replicate mxl Err));
        result = lbvjvm G mxs (1+size pTs+mxl) rT et cert ins (OK start)
    in result ≠ Err)"

definition wt_jvm_prog_lbv :: "jvm_prog ⇒ prog_cert ⇒ bool" where
  "wt_jvm_prog_lbv G cert ≡
  wf_prog (λG C (sig,rT,(maxs,maxl,b,et)). wt_lbv G C (snd sig) rT maxs maxl et (cert
C sig) b) G"

definition mk_cert :: "jvm_prog ⇒ nat ⇒ ty ⇒ exception_table ⇒ instr list
             ⇒ method_type ⇒ JVMType.state list" where
  "mk_cert G maxs rT et bs phi ≡ make_cert (exec G maxs rT et bs) (map OK phi) (OK None)"

definition prg_cert :: "jvm_prog ⇒ prog_type ⇒ prog_cert" where
  "prg_cert G phi C sig ≡ let (C,rT,(maxs,maxl,ins,et)) = the (method (G,C) sig) in
                           mk_cert G maxs rT et ins (phi C sig)"


lemma wt_method_def2:
  fixes pTs and mxl and G and mxs and rT and et and bs and phi
  defines [simp]: "mxr   ≡ 1 + length pTs + mxl"
  defines [simp]: "r     ≡ sup_state_opt G"
  defines [simp]: "app0  ≡ λpc. app (bs!pc) G mxs rT pc et"
  defines [simp]: "step0 ≡ λpc. eff (bs!pc) G pc et"

  shows
  "wt_method G C pTs rT mxs mxl bs et phi =
  (bs ≠ [] ∧
   length phi = length bs ∧
   check_bounded bs et ∧
   check_types G mxs mxr (map OK phi) ∧
   wt_start G C pTs mxl phi ∧
   wt_app_eff r app0 step0 phi)"
  ⟨proof⟩


lemma check_certD:
  "check_cert G mxs mxr n cert ⟹ cert_ok cert n Err (OK None) (states G mxs mxr)"
  ⟨proof⟩


lemma wt_lbv_wt_step:
  assumes wf:  "wf_prog wf_mb G"
  assumes lbv: "wt_lbv G C pTs rT mxs mxl et cert ins"
  assumes C:   "is_class G C"
  assumes pTs: "set pTs ⊆ types G"
```

defines *[simp]: "mxr ≡ 1+length pTs+mxl"*

shows *"∃ts ∈ list (size ins) (states G mxs mxr).*
wt_step (JVMType.le G mxs mxr) Err (exec G mxs rT et ins) ts
∧ OK (Some ([],(OK (Class C))#((map OK pTs))@(replicate mxl Err))) <=_(JVMType.le G mxs mxr) ts!0"
⟨*proof*⟩

lemma *wt_lbv_wt_method:*
assumes *wf: "wf_prog wf_mb G"*
assumes *lbv: "wt_lbv G C pTs rT mxs mxl et cert ins"*
assumes *C: "is_class G C"*
assumes *pTs: "set pTs ⊆ types G"*

shows *"∃phi. wt_method G C pTs rT mxs mxl ins et phi"*
⟨*proof*⟩

lemma *wt_method_wt_lbv:*
assumes *wf: "wf_prog wf_mb G"*
assumes *wt: "wt_method G C pTs rT mxs mxl ins et phi"*
assumes *C: "is_class G C"*
assumes *pTs: "set pTs ⊆ types G"*

defines *[simp]: "cert ≡ mk_cert G mxs rT et ins phi"*

shows *"wt_lbv G C pTs rT mxs mxl et cert ins"*
⟨*proof*⟩

theorem *jvm_lbv_correct:*
"wt_jvm_prog_lbv G Cert ⟹ ∃Phi. wt_jvm_prog G Phi"
⟨*proof*⟩

theorem *jvm_lbv_complete:*
"wt_jvm_prog G Phi ⟹ wt_jvm_prog_lbv G (prg_cert G Phi)"
⟨*proof*⟩

end

4.22 BV Type Safety Invariant

theory *Correct*
imports *BVSpec "../JVM/JVMExec"*
begin

definition *approx_val :: "[jvm_prog,aheap,val,ty err] ⇒ bool"* **where**
"approx_val G h v any == case any of Err ⇒ True | OK T ⇒ G,h⊢v::≼T"

definition *approx_loc :: "[jvm_prog,aheap,val list,locvars_type] ⇒ bool"* **where**
"approx_loc G hp loc LT == list_all2 (approx_val G hp) loc LT"

definition *approx_stk :: "[jvm_prog,aheap,opstack,opstack_type] ⇒ bool"* **where**
"approx_stk G hp stk ST == approx_loc G hp stk (map OK ST)"

definition *correct_frame :: "[jvm_prog,aheap,state_type,nat,bytecode] ⇒ frame ⇒ bool"* **where**
"correct_frame G hp == λ(ST,LT) maxl ins (stk,loc,C,sig,pc).
approx_stk G hp stk ST ∧ approx_loc G hp loc LT ∧
pc < length ins ∧ length loc=length(snd sig)+maxl+1"

primrec *correct_frames :: "[jvm_prog,aheap,prog_type,ty,sig,frame list] ⇒ bool"* **where**
"correct_frames G hp phi rT0 sig0 [] = True"
| "correct_frames G hp phi rT0 sig0 (f#frs) =
(let (stk,loc,C,sig,pc) = f in
(∃ST LT rT maxs maxl ins et.
phi C sig ! pc = Some (ST,LT) ∧ is_class G C ∧
method (G,C) sig = Some(C,rT,(maxs,maxl,ins,et)) ∧
(∃C' mn pTs. ins!pc = (Invoke C' mn pTs) ∧
(mn,pTs) = sig0 ∧
(∃apTs D ST' LT'.
(phi C sig)!pc = Some ((rev apTs) @ (Class D) # ST', LT') ∧
length apTs = length pTs ∧
(∃D' rT' maxs' maxl' ins' et'.
method (G,D) sig0 = Some(D',rT',(maxs',maxl',ins',et')) ∧
G ⊢ rT0 ≼ rT') ∧
correct_frame G hp (ST, LT) maxl ins f ∧
correct_frames G hp phi rT sig frs))))"

definition *correct_state :: "[jvm_prog,prog_type,jvm_state] ⇒ bool"*
(⟨_,_ ⊢JVM _ √⟩ [51,51] 50) **where**
"correct_state G phi == λ(xp,hp,frs).
case xp of
None ⇒ (case frs of
[] ⇒ True
| (f#fs) ⇒ G⊢h hp√ ∧ preallocated hp ∧
(let (stk,loc,C,sig,pc) = f
in
∃rT maxs maxl ins et s.
is_class G C ∧
method (G,C) sig = Some(C,rT,(maxs,maxl,ins,et)) ∧
phi C sig ! pc = Some s ∧
correct_frame G hp s maxl ins f ∧
correct_frames G hp phi rT sig fs))
| Some x ⇒ frs = []"

lemma *sup_ty_opt_OK:*
"(G ⊢ X <=o (OK T')) = (∃T. X = OK T ∧ G ⊢ T ≼ T')"
⟨*proof*⟩

4.22.1 approx-val

lemma *approx_val_Err [simp,intro!]:*
"approx_val G hp x Err"

⟨*proof*⟩

lemma *approx_val_OK [iff]:*
"approx_val G hp x (OK T) = (G,hp ⊢ x ::≼ T)"
⟨*proof*⟩

lemma *approx_val_Null [simp,intro!]:*
"approx_val G hp Null (OK (RefT x))"
⟨*proof*⟩

lemma *approx_val_sup_heap:*
"⟦ approx_val G hp v T; hp ≤| hp' ⟧ ⟹ approx_val G hp' v T"
⟨*proof*⟩

lemma *approx_val_heap_update:*
"⟦ hp a = Some obj'; G,hp⊢ v::≼T; obj_ty obj = obj_ty obj'⟧
⟹ G,hp(a↦obj)⊢ v::≼T"
⟨*proof*⟩

lemma *approx_val_widen:*
"⟦ approx_val G hp v T; G ⊢ T <=o T'; wf_prog wt G ⟧
⟹ approx_val G hp v T'"
⟨*proof*⟩

4.22.2 approx-loc

lemma *approx_loc_Nil [simp,intro!]:*
"approx_loc G hp [] []"
⟨*proof*⟩

lemma *approx_loc_Cons [iff]:*
"approx_loc G hp (l#ls) (L#LT) =
(approx_val G hp l L ∧ approx_loc G hp ls LT)"
⟨*proof*⟩

lemma *approx_loc_nth:*
"⟦ approx_loc G hp loc LT; n < length LT ⟧
⟹ approx_val G hp (loc!n) (LT!n)"
⟨*proof*⟩

lemma *approx_loc_imp_approx_val_sup:*
"⟦approx_loc G hp loc LT; n < length LT; LT ! n = OK T; G ⊢ T ≼ T'; wf_prog wt G⟧
⟹ G,hp ⊢ (loc!n) ::≼ T'"
⟨*proof*⟩

lemma *approx_loc_conv_all_nth:*
"approx_loc G hp loc LT =
(length loc = length LT ∧ (∀n < length loc. approx_val G hp (loc!n) (LT!n)))"
⟨*proof*⟩

lemma *approx_loc_sup_heap:*
"⟦ approx_loc G hp loc LT; hp ≤| hp' ⟧
⟹ approx_loc G hp' loc LT"
⟨*proof*⟩

lemma *approx_loc_widen:*
```
"⟦ approx_loc G hp loc LT; G ⊢ LT <=l LT'; wf_prog wt G ⟧
⟹ approx_loc G hp loc LT'"
```
⟨*proof*⟩

lemma *loc_widen_Err [dest]:*
```
"⋀XT. G ⊢ replicate n Err <=l XT ⟹ XT = replicate n Err"
```
⟨*proof*⟩

lemma *approx_loc_Err [iff]:*
```
"approx_loc G hp (replicate n v) (replicate n Err)"
```
⟨*proof*⟩

lemma *approx_loc_subst:*
```
"⟦ approx_loc G hp loc LT; approx_val G hp x X ⟧
⟹ approx_loc G hp (loc[idx:=x]) (LT[idx:=X])"
```
⟨*proof*⟩

lemma *approx_loc_append:*
```
"length l1=length L1 ⟹
approx_loc G hp (l1@l2) (L1@L2) =
(approx_loc G hp l1 L1 ∧ approx_loc G hp l2 L2)"
```
⟨*proof*⟩

4.22.3 approx-stk

lemma *approx_stk_rev_lem:*
```
"approx_stk G hp (rev s) (rev t) = approx_stk G hp s t"
```
⟨*proof*⟩

lemma *approx_stk_rev:*
```
"approx_stk G hp (rev s) t = approx_stk G hp s (rev t)"
```
⟨*proof*⟩

lemma *approx_stk_sup_heap:*
```
"⟦ approx_stk G hp stk ST; hp ≤| hp' ⟧ ⟹ approx_stk G hp' stk ST"
```
⟨*proof*⟩

lemma *approx_stk_widen:*
```
"⟦ approx_stk G hp stk ST; G ⊢ map OK ST <=l map OK ST'; wf_prog wt G ⟧
⟹ approx_stk G hp stk ST'"
```
⟨*proof*⟩

lemma *approx_stk_Nil [iff]:*
```
"approx_stk G hp [] []"
```
⟨*proof*⟩

lemma *approx_stk_Cons [iff]:*
```
"approx_stk G hp (x#stk) (S#ST) =
(approx_val G hp x (OK S) ∧ approx_stk G hp stk ST)"
```
⟨*proof*⟩

lemma *approx_stk_Cons_lemma [iff]:*

```
  "approx_stk G hp stk (S#ST') =
  (∃s stk'. stk = s#stk' ∧ approx_val G hp s (OK S) ∧ approx_stk G hp stk' ST')"
```
⟨*proof*⟩

lemma *approx_stk_append:*
```
  "approx_stk G hp stk (S@S') ⟹
  (∃s stk'. stk = s@stk' ∧ length s = length S ∧ length stk' = length S' ∧
            approx_stk G hp s S ∧ approx_stk G hp stk' S')"
```
⟨*proof*⟩

lemma *approx_stk_all_widen:*
```
  "⟦ approx_stk G hp stk ST; ∀(x, y) ∈ set (zip ST ST'). G ⊢ x ≼ y; length ST = length
ST'; wf_prog wt G ⟧
  ⟹ approx_stk G hp stk ST'"
```
⟨*proof*⟩

4.22.4 oconf

lemma *oconf_field_update:*
```
  "⟦map_of (fields (G, oT)) FD = Some T; G,hp⊢v::≼T; G,hp⊢(oT,fs)√ ⟧
  ⟹ G,hp⊢(oT, fs(FD↦v))√"
```
⟨*proof*⟩

lemma *oconf_newref:*
```
  "⟦hp oref = None; G,hp ⊢ obj √; G,hp ⊢ obj' √⟧ ⟹ G,hp(oref↦obj') ⊢ obj √"
```
⟨*proof*⟩

lemma *oconf_heap_update:*
```
  "⟦ hp a = Some obj'; obj_ty obj' = obj_ty obj''; G,hp⊢obj√ ⟧
  ⟹ G,hp(a↦obj'')⊢obj√"
```
⟨*proof*⟩

4.22.5 hconf

lemma *hconf_newref:*
```
  "⟦ hp oref = None; G⊢h hp√; G,hp⊢obj√ ⟧ ⟹ G⊢h hp(oref↦obj)√"
```
⟨*proof*⟩

lemma *hconf_field_update:*
```
  "⟦ map_of (fields (G, oT)) X = Some T; hp a = Some(oT,fs);
     G,hp⊢v::≼T; G⊢h hp√ ⟧
  ⟹ G⊢h hp(a ↦ (oT, fs(X↦v)))√"
```
⟨*proof*⟩

4.22.6 preallocated

lemma *preallocated_field_update:*
```
  "⟦ map_of (fields (G, oT)) X = Some T; hp a = Some(oT,fs);
     G⊢h hp√; preallocated hp ⟧
  ⟹ preallocated (hp(a ↦ (oT, fs(X↦v))))"
```
⟨*proof*⟩

lemma
assumes *none:* `"hp oref = None"` **and** *alloc:* `"preallocated hp"`

shows *preallocated_newref: "preallocated (hp(oref↦obj))"*
⟨*proof*⟩

4.22.7 correct-frames

lemmas *[simp del] = fun_upd_apply*

lemma *correct_frames_field_update [rule_format]:*
"∀rT C sig.
correct_frames G hp phi rT sig frs ⟶
hp a = Some (C,fs) ⟶
map_of (fields (G, C)) fl = Some fd ⟶
G,hp⊢v::≼fd
⟶ correct_frames G (hp(a ↦ (C, fs(fl↦v)))) phi rT sig frs"
⟨*proof*⟩

lemma *correct_frames_newref [rule_format]:*
"∀rT C sig.
hp x = None ⟶
correct_frames G hp phi rT sig frs ⟶
correct_frames G (hp(x ↦ obj)) phi rT sig frs"
⟨*proof*⟩

end

4.23 BV Type Safety Proof

theory *BVSpecTypeSafe*
imports *Correct*
begin

This theory contains proof that the specification of the bytecode verifier only admits type safe programs.

4.23.1 Preliminaries

Simp and intro setup for the type safety proof:

lemmas *defs1 = sup_state_conv correct_state_def correct_frame_def*
wt_instr_def eff_def norm_eff_def

lemmas *widen_rules[intro] = approx_val_widen approx_loc_widen approx_stk_widen*

lemmas *[simp del] = split_paired_All*

If we have a welltyped program and a conforming state, we can directly infer that the current instruction is well typed:

lemma *wt_jvm_prog_impl_wt_instr_cor:*
"⟦ wt_jvm_prog G phi; method (G,C) sig = Some (C,rT,maxs,maxl,ins,et);
G,phi ⊢JVM (None, hp, (stk,loc,C,sig,pc)#frs)√ ⟧
⟹ wt_instr (ins!pc) G rT (phi C sig) maxs (length ins) et pc"
⟨*proof*⟩

4.23.2 Exception Handling

Exceptions don't touch anything except the stack:

lemma *exec_instr_xcpt:*
```
"(fst (exec_instr i G hp stk vars Cl sig pc frs) = Some xcp)
= (∃stk'. exec_instr i G hp stk vars Cl sig pc frs =
          (Some xcp, hp, (stk', vars, Cl, sig, pc)#frs))"
```
⟨*proof*⟩

Relates *match_any* from the Bytecode Verifier with *match_exception_table* from the operational semantics:

lemma *in_match_any:*
```
"match_exception_table G xcpt pc et = Some pc' ⟹
∃C. C ∈ set (match_any G pc et) ∧ G ⊢ xcpt ⪯C C ∧
    match_exception_table G C pc et = Some pc'"
(is "PROP ?P et" is "?match et ⟹ ?match_any et")
```
⟨*proof*⟩

lemma *match_et_imp_match:*
```
"match_exception_table G (Xcpt X) pc et = Some handler
⟹ match G X pc et = [Xcpt X]"
```
⟨*proof*⟩

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

lemma *uncaught_xcpt_correct:*
```
"⋀f. ⟦ wt_jvm_prog G phi; xcp = Addr adr; hp adr = Some T;
    G,phi ⊢JVM (None, hp, f#frs)√ ⟧
⟹ G,phi ⊢JVM (find_handler G (Some xcp) hp frs)√"
(is "⋀f. ⟦ ?wt; ?adr; ?hp; ?correct (None, hp, f#frs) ⟧ ⟹ ?correct (?find frs)")
```
⟨*proof*⟩

declare *raise_if_def [simp]*

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

lemma *exec_instr_xcpt_hp:*
```
"⟦ fst (exec_instr (ins!pc) G hp stk vars Cl sig pc frs) = Some xcp;
   wt_instr (ins!pc) G rT (phi C sig) maxs (length ins) et pc;
   G,phi ⊢JVM (None, hp, (stk,loc,C,sig,pc)#frs)√ ⟧
⟹ ∃adr T. xcp = Addr adr ∧ hp adr = Some T"
(is "⟦ ?xcpt; ?wt; ?correct ⟧ ⟹ ?thesis")
```
⟨*proof*⟩

lemma *cname_of_xcp [intro]:*
```
"⟦preallocated hp; xcp = Addr (XcptRef x)⟧ ⟹ cname_of hp xcp = Xcpt x"
```
⟨*proof*⟩

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

lemma *xcpt_correct:*
```
  "⟦ wt_jvm_prog G phi;
      method (G,C) sig = Some (C,rT,maxs,maxl,ins,et);
      wt_instr (ins!pc) G rT (phi C sig) maxs (length ins) et pc;
      fst (exec_instr (ins!pc) G hp stk loc C sig pc frs) = Some xcp;
      Some state' = exec (G, None, hp, (stk,loc,C,sig,pc)#frs);
      G,phi ⊢JVM (None, hp, (stk,loc,C,sig,pc)#frs)√ ⟧
  ⟹ G,phi ⊢JVM state'√"
```
⟨*proof*⟩

4.23.3 Single Instructions

In this section we look at each single (welltyped) instruction, and prove that the state after execution of the instruction still conforms. Since we have already handled exceptions above, we can now assume, that on exception occurs for this (single step) execution.

lemmas *[iff] = not_Err_eq*

lemma *Load_correct:*
```
"⟦ wf_prog wt G;
    method (G,C) sig = Some (C,rT,maxs,maxl,ins,et);
    ins!pc = Load idx;
    wt_instr (ins!pc) G rT (phi C sig) maxs (length ins) et pc;
    Some state' = exec (G, None, hp, (stk,loc,C,sig,pc)#frs);
    G,phi ⊢JVM (None, hp, (stk,loc,C,sig,pc)#frs)√ ⟧
⟹ G,phi ⊢JVM state'√"
```
⟨*proof*⟩

lemma *Store_correct:*
```
"⟦ wf_prog wt G;
  method (G,C) sig = Some (C,rT,maxs,maxl,ins,et);
  ins!pc = Store idx;
  wt_instr (ins!pc) G rT (phi C sig) maxs (length ins) et pc;
  Some state' = exec (G, None, hp, (stk,loc,C,sig,pc)#frs);
  G,phi ⊢JVM (None, hp, (stk,loc,C,sig,pc)#frs)√ ⟧
⟹ G,phi ⊢JVM state'√"
```
⟨*proof*⟩

lemma *LitPush_correct:*
```
"⟦ wf_prog wt G;
    method (G,C) sig = Some (C,rT,maxs,maxl,ins,et);
    ins!pc = LitPush v;
    wt_instr (ins!pc) G rT (phi C sig) maxs (length ins) et pc;
    Some state' = exec (G, None, hp, (stk,loc,C,sig,pc)#frs);
    G,phi ⊢JVM (None, hp, (stk,loc,C,sig,pc)#frs)√ ⟧
⟹ G,phi ⊢JVM state'√"
```
⟨*proof*⟩

lemma *Cast_conf2:*
```
  "⟦ wf_prog ok G; G,h⊢v::≼RefT rt; cast_ok G C h v;
      G⊢Class C≼T; is_class G C⟧
  ⟹ G,h⊢v::≼T"
```

⟨*proof*⟩

lemmas *defs2 = defs1 raise_system_xcpt_def*

lemma *Checkcast_correct:*
```
"⟦ wt_jvm_prog G phi;
    method (G,C) sig = Some (C,rT,maxs,maxl,ins,et);
    ins!pc = Checkcast D;
    wt_instr (ins!pc) G rT (phi C sig) maxs (length ins) et pc;
    Some state' = exec (G, None, hp, (stk,loc,C,sig,pc)#frs) ;
    G,phi ⊢JVM (None, hp, (stk,loc,C,sig,pc)#frs)√;
    fst (exec_instr (ins!pc) G hp stk loc C sig pc frs) = None ⟧
⟹ G,phi ⊢JVM state'√"
```
⟨*proof*⟩

lemma *Getfield_correct:*
```
"⟦ wt_jvm_prog G phi;
  method (G,C) sig = Some (C,rT,maxs,maxl,ins,et);
  ins!pc = Getfield F D;
  wt_instr (ins!pc) G rT (phi C sig) maxs (length ins) et pc;
  Some state' = exec (G, None, hp, (stk,loc,C,sig,pc)#frs) ;
  G,phi ⊢JVM (None, hp, (stk,loc,C,sig,pc)#frs)√;
  fst (exec_instr (ins!pc) G hp stk loc C sig pc frs) = None ⟧
⟹ G,phi ⊢JVM state'√"
```
⟨*proof*⟩

lemma *Putfield_correct:*
```
"⟦ wf_prog wt G;
  method (G,C) sig = Some (C,rT,maxs,maxl,ins,et);
  ins!pc = Putfield F D;
  wt_instr (ins!pc) G rT (phi C sig) maxs (length ins) et pc;
  Some state' = exec (G, None, hp, (stk,loc,C,sig,pc)#frs) ;
  G,phi ⊢JVM (None, hp, (stk,loc,C,sig,pc)#frs)√;
  fst (exec_instr (ins!pc) G hp stk loc C sig pc frs) = None ⟧
⟹ G,phi ⊢JVM state'√"
```
⟨*proof*⟩

lemma *New_correct:*
```
"⟦ wf_prog wt G;
  method (G,C) sig = Some (C,rT,maxs,maxl,ins,et);
  ins!pc = New X;
  wt_instr (ins!pc) G rT (phi C sig) maxs (length ins) et pc;
  Some state' = exec (G, None, hp, (stk,loc,C,sig,pc)#frs) ;
  G,phi ⊢JVM (None, hp, (stk,loc,C,sig,pc)#frs)√;
  fst (exec_instr (ins!pc) G hp stk loc C sig pc frs) = None ⟧
⟹ G,phi ⊢JVM state'√"
```
⟨*proof*⟩

lemmas *[simp del] = split_paired_Ex*

lemma *Invoke_correct:*
"⟦ *wt_jvm_prog G phi;*
method (G,C) sig = Some (C,rT,maxs,maxl,ins,et);
ins ! pc = Invoke C' mn pTs;
wt_instr (ins!pc) G rT (phi C sig) maxs (length ins) et pc;
Some state' = exec (G, None, hp, (stk,loc,C,sig,pc)#frs) ;
G,phi ⊢JVM (None, hp, (stk,loc,C,sig,pc)#frs)√;
fst (exec_instr (ins!pc) G hp stk loc C sig pc frs) = None ⟧
⟹ *G,phi ⊢JVM state'√*"
⟨*proof*⟩

lemmas *[simp del] = map_append*

lemma *Return_correct:*
"⟦ *wt_jvm_prog G phi;*
method (G,C) sig = Some (C,rT,maxs,maxl,ins,et);
ins ! pc = Return;
wt_instr (ins!pc) G rT (phi C sig) maxs (length ins) et pc;
Some state' = exec (G, None, hp, (stk,loc,C,sig,pc)#frs) ;
G,phi ⊢JVM (None, hp, (stk,loc,C,sig,pc)#frs)√ ⟧
⟹ *G,phi ⊢JVM state'√*"
⟨*proof*⟩

lemmas *[simp] = map_append*

lemma *Goto_correct:*
"⟦ *wf_prog wt G;*
method (G,C) sig = Some (C,rT,maxs,maxl,ins,et);
ins ! pc = Goto branch;
wt_instr (ins!pc) G rT (phi C sig) maxs (length ins) et pc;
Some state' = exec (G, None, hp, (stk,loc,C,sig,pc)#frs) ;
G,phi ⊢JVM (None, hp, (stk,loc,C,sig,pc)#frs)√ ⟧
⟹ *G,phi ⊢JVM state'√*"
⟨*proof*⟩

lemma *Ifcmpeq_correct:*
"⟦ *wf_prog wt G;*
method (G,C) sig = Some (C,rT,maxs,maxl,ins,et);
ins ! pc = Ifcmpeq branch;
wt_instr (ins!pc) G rT (phi C sig) maxs (length ins) et pc;
Some state' = exec (G, None, hp, (stk,loc,C,sig,pc)#frs) ;
G,phi ⊢JVM (None, hp, (stk,loc,C,sig,pc)#frs)√ ⟧
⟹ *G,phi ⊢JVM state'√*"
⟨*proof*⟩

lemma *Pop_correct:*
"⟦ *wf_prog wt G;*
method (G,C) sig = Some (C,rT,maxs,maxl,ins,et);
ins ! pc = Pop;
wt_instr (ins!pc) G rT (phi C sig) maxs (length ins) et pc;
Some state' = exec (G, None, hp, (stk,loc,C,sig,pc)#frs) ;
G,phi ⊢JVM (None, hp, (stk,loc,C,sig,pc)#frs)√ ⟧
⟹ *G,phi ⊢JVM state'√*"

⟨*proof*⟩

lemma *Dup_correct:*
```
"⟦ wf_prog wt G;
  method (G,C) sig = Some (C,rT,maxs,maxl,ins,et);
  ins ! pc = Dup;
  wt_instr (ins!pc) G rT (phi C sig) maxs (length ins) et pc;
  Some state' = exec (G, None, hp, (stk,loc,C,sig,pc)#frs) ;
  G,phi ⊢JVM (None, hp, (stk,loc,C,sig,pc)#frs)√ ⟧
⟹ G,phi ⊢JVM state'√"
```
⟨*proof*⟩

lemma *Dup_x1_correct:*
```
"⟦ wf_prog wt G;
  method (G,C) sig = Some (C,rT,maxs,maxl,ins,et);
  ins ! pc = Dup_x1;
  wt_instr (ins!pc) G rT (phi C sig) maxs (length ins) et pc;
  Some state' = exec (G, None, hp, (stk,loc,C,sig,pc)#frs) ;
  G,phi ⊢JVM (None, hp, (stk,loc,C,sig,pc)#frs)√ ⟧
⟹ G,phi ⊢JVM state'√"
```
⟨*proof*⟩

lemma *Dup_x2_correct:*
```
"⟦ wf_prog wt G;
  method (G,C) sig = Some (C,rT,maxs,maxl,ins,et);
  ins ! pc = Dup_x2;
  wt_instr (ins!pc) G rT (phi C sig) maxs (length ins) et pc;
  Some state' = exec (G, None, hp, (stk,loc,C,sig,pc)#frs) ;
  G,phi ⊢JVM (None, hp, (stk,loc,C,sig,pc)#frs)√ ⟧
⟹ G,phi ⊢JVM state'√"
```
⟨*proof*⟩

lemma *Swap_correct:*
```
"⟦ wf_prog wt G;
  method (G,C) sig = Some (C,rT,maxs,maxl,ins,et);
  ins ! pc = Swap;
  wt_instr (ins!pc) G rT (phi C sig) maxs (length ins) et pc;
  Some state' = exec (G, None, hp, (stk,loc,C,sig,pc)#frs) ;
  G,phi ⊢JVM (None, hp, (stk,loc,C,sig,pc)#frs)√ ⟧
⟹ G,phi ⊢JVM state'√"
```
⟨*proof*⟩

lemma *IAdd_correct:*
```
"⟦ wf_prog wt G;
  method (G,C) sig = Some (C,rT,maxs,maxl,ins,et);
  ins ! pc = IAdd;
  wt_instr (ins!pc) G rT (phi C sig) maxs (length ins) et pc;
  Some state' = exec (G, None, hp, (stk,loc,C,sig,pc)#frs) ;
  G,phi ⊢JVM (None, hp, (stk,loc,C,sig,pc)#frs)√ ⟧
⟹ G,phi ⊢JVM state'√"
```
⟨*proof*⟩

lemma *Throw_correct:*
```
"⟦ wf_prog wt G;
```

```
  method (G,C) sig = Some (C,rT,maxs,maxl,ins,et);
  ins ! pc = Throw;
  Some state' = exec (G, None, hp, (stk,loc,C,sig,pc)#frs) ;
  G,phi ⊢JVM (None, hp, (stk,loc,C,sig,pc)#frs)√;
  fst (exec_instr (ins!pc) G hp stk loc C sig pc frs) = None ⟧
⟹ G,phi ⊢JVM state'√"
```
⟨*proof*⟩

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 well-typed programs, a conforming state is transformed into another conforming state when one instruction is executed.

theorem *instr_correct:*
```
"⟦ wt_jvm_prog G phi;
  method (G,C) sig = Some (C,rT,maxs,maxl,ins,et);
  Some state' = exec (G, None, hp, (stk,loc,C,sig,pc)#frs);
  G,phi ⊢JVM (None, hp, (stk,loc,C,sig,pc)#frs)√ ⟧
⟹ G,phi ⊢JVM state'√"
```
⟨*proof*⟩

4.23.4 Main

lemma *correct_state_impl_Some_method:*
```
  "G,phi ⊢JVM (None, hp, (stk,loc,C,sig,pc)#frs)√
  ⟹ ∃meth. method (G,C) sig = Some(C,meth)"
```
⟨*proof*⟩

lemma *BV_correct_1 [rule_format]:*
```
"⋀state. ⟦ wt_jvm_prog G phi; G,phi ⊢JVM state√⟧
 ⟹ exec (G,state) = Some state' ⟶ G,phi ⊢JVM state'√"
```
⟨*proof*⟩

lemma *L0:*
```
  "⟦ xp=None; frs≠[] ⟧ ⟹ (∃state'. exec (G,xp,hp,frs) = Some state')"
```
⟨*proof*⟩

lemma *L1:*
```
  "⟦wt_jvm_prog G phi; G,phi ⊢JVM (xp,hp,frs)√; xp=None; frs≠[]⟧
  ⟹ ∃state'. exec(G,xp,hp,frs) = Some state' ∧ G,phi ⊢JVM state'√"
```
⟨*proof*⟩

theorem *BV_correct [rule_format]:*
```
"⟦ wt_jvm_prog G phi; G ⊢ s −jvm→ t ⟧ ⟹ G,phi ⊢JVM s√ ⟶ G,phi ⊢JVM t√"
```
⟨*proof*⟩

theorem *BV_correct_implies_approx:*
```
"⟦ wt_jvm_prog G phi;
    G ⊢ s0 −jvm→ (None,hp,(stk,loc,C,sig,pc)#frs); G,phi ⊢JVM s0 √⟧
⟹ approx_stk G hp stk (fst (the (phi C sig ! pc))) ∧
    approx_loc G hp loc (snd (the (phi C sig ! pc)))"
```

⟨*proof*⟩

lemma
fixes *G :: jvm_prog (‹Γ›)*
assumes *wf: "wf_prog wf_mb Γ"*
shows *hconf_start: "Γ ⊢h (start_heap Γ) √"*
⟨*proof*⟩

lemma
fixes *G :: jvm_prog (‹Γ›)* **and** *Phi :: prog_type (‹Φ›)*
shows *BV_correct_initial:*
"wt_jvm_prog Γ Φ ⟹ is_class Γ C ⟹ method (Γ,C) (m,[]) = Some (C, b)
⟹ Γ,Φ ⊢JVM start_state G C m √"
⟨*proof*⟩

theorem *typesafe:*
fixes *G :: jvm_prog (‹Γ›)*
and *Phi :: prog_type (‹Φ›)*
assumes *welltyped: "wt_jvm_prog Γ Φ"*
and *main_method: "is_class Γ C" "method (Γ,C) (m,[]) = Some (C, b)"*
and *exec_all: "G ⊢ start_state Γ C m −jvm→ s"*
shows *"Γ,Φ ⊢JVM s √"*
⟨*proof*⟩

end

4.24 Welltyped Programs produce no Type Errors

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

Some simple lemmas about the type testing functions of the defensive JVM:

lemma *typeof_NoneD [simp,dest]:*
"typeof (λv. None) v = Some x ⟹ ¬isAddr v"
⟨*proof*⟩

lemma *isRef_def2:*
"isRef v = (v = Null ∨ (∃loc. v = Addr loc))"
⟨*proof*⟩

lemma *app'Store[simp]:*
"app' (Store idx, G, pc, maxs, rT, (ST,LT)) = (∃T ST'. ST = T#ST' ∧ idx < length LT)"
⟨*proof*⟩

lemma *app'GetField[simp]:*
"app' (Getfield F C, G, pc, maxs, rT, (ST,LT)) =
(∃oT vT ST'. ST = oT#ST' ∧ is_class G C ∧
field (G,C) F = Some (C,vT) ∧ G ⊢ oT ≼ Class C)"
⟨*proof*⟩

lemma *app'PutField[simp]:*
"app' (Putfield F C, G, pc, maxs, rT, (ST,LT)) =

```
 (∃vT vT' oT ST'. ST = vT#oT#ST' ∧ is_class G C ∧
  field (G,C) F = Some (C, vT') ∧
  G ⊢ oT ≼ Class C ∧ G ⊢ vT ≼ vT')"
```
⟨*proof*⟩

lemma *app'Checkcast[simp]:*
```
"app' (Checkcast C, G, pc, maxs, rT, (ST,LT)) =
 (∃rT ST'. ST = RefT rT#ST' ∧ is_class G C)"
```
⟨*proof*⟩

lemma *app'Pop[simp]:*
```
"app' (Pop, G, pc, maxs, rT, (ST,LT)) = (∃T ST'. ST = T#ST')"
```
⟨*proof*⟩

lemma *app'Dup[simp]:*
```
"app' (Dup, G, pc, maxs, rT, (ST,LT)) =
(∃T ST'. ST = T#ST' ∧ length ST < maxs)"
```
⟨*proof*⟩

lemma *app'Dup_x1[simp]:*
```
"app' (Dup_x1, G, pc, maxs, rT, (ST,LT)) =
(∃T1 T2 ST'. ST = T1#T2#ST' ∧ length ST < maxs)"
```
⟨*proof*⟩

lemma *app'Dup_x2[simp]:*
```
"app' (Dup_x2, G, pc, maxs, rT, (ST,LT)) =
(∃T1 T2 T3 ST'. ST = T1#T2#T3#ST' ∧ length ST < maxs)"
```
⟨*proof*⟩

lemma *app'Swap[simp]:*
```
"app' (Swap, G, pc, maxs, rT, (ST,LT)) = (∃T1 T2 ST'. ST = T1#T2#ST')"
```
⟨*proof*⟩

lemma *app'IAdd[simp]:*
```
"app' (IAdd, G, pc, maxs, rT, (ST,LT)) =
(∃ST'. ST = PrimT Integer#PrimT Integer#ST')"
```
⟨*proof*⟩

lemma *app'Ifcmpeq[simp]:*
```
"app' (Ifcmpeq b, G, pc, maxs, rT, (ST,LT)) =
(∃T1 T2 ST'. ST = T1#T2#ST' ∧ 0 ≤ b + int pc ∧
((∃p. T1 = PrimT p ∧ T1 = T2) ∨
(∃r r'. T1 = RefT r ∧ T2 = RefT r')))"
```
⟨*proof*⟩

lemma *app'Return[simp]:*

```
  "app' (Return, G, pc, maxs, rT, (ST,LT)) =
  (∃T ST'. ST = T#ST'∧ G ⊢ T ⪯ rT)"
```
⟨*proof*⟩

lemma *app'Throw[simp]:*
```
  "app' (Throw, G, pc, maxs, rT, (ST,LT)) =
  (∃ST' r. ST = RefT r#ST')"
```
⟨*proof*⟩

lemma *app'Invoke[simp]:*
```
"app' (Invoke C mn fpTs, G, pc, maxs, rT, ST, LT) =
 (∃apTs X ST' mD' rT' b'.
  ST = (rev apTs) @ X # ST' ∧
  length apTs = length fpTs ∧ is_class G C ∧
  (∀(aT,fT)∈set(zip apTs fpTs). G ⊢ aT ⪯ fT) ∧
  method (G,C) (mn,fpTs) = Some (mD', rT', b') ∧ G ⊢ X ⪯ Class C)"
  (is "?app ST LT = ?P ST LT")
```
⟨*proof*⟩

lemma *approx_loc_len [simp]:*
```
  "approx_loc G hp loc LT ⟹ length loc = length LT"
```
⟨*proof*⟩

lemma *approx_stk_len [simp]:*
```
  "approx_stk G hp stk ST ⟹ length stk = length ST"
```
⟨*proof*⟩

lemma *isRefI [intro, simp]:* `"G,hp ⊢ v ::⪯ RefT T ⟹ isRef v"`
⟨*proof*⟩

lemma *isIntgI [intro, simp]:* `"G,hp ⊢ v ::⪯ PrimT Integer ⟹ isIntg v"`
⟨*proof*⟩

lemma *list_all2_approx:*
```
  "list_all2 (approx_val G hp) s (map OK S) = list_all2 (conf G hp) s S"
```
⟨*proof*⟩

lemma *list_all2_conf_widen:*
```
  "wf_prog mb G ⟹
  list_all2 (conf G hp) a b ⟹
  list_all2 (λx y. G ⊢ x ⪯ y) b c ⟹
  list_all2 (conf G hp) a c"
```
⟨*proof*⟩

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

theorem *no_type_error:*
assumes *welltyped:* `"wt_jvm_prog G Phi"` **and** *conforms:* `"G,Phi ⊢JVM s √"`
shows `"exec_d G (Normal s) ≠ TypeError"`
⟨*proof*⟩

The theorem above tells us that, in welltyped programs, the defensive machine reaches the

same result as the aggressive one (after arbitrarily many steps).

theorem *welltyped_aggressive_imp_defensive:*
"wt_jvm_prog G Phi ⟹ G,Phi ⊢JVM s √ ⟹ G ⊢ s −jvm→ t
⟹ G ⊢ (Normal s) −jvmd→ (Normal t)"
⟨*proof*⟩

lemma *neq_TypeError_eq [simp]: "s ≠ TypeError = (∃s'. s = Normal s')"*
⟨*proof*⟩

theorem *no_type_errors:*
"wt_jvm_prog G Phi ⟹ G,Phi ⊢JVM s √
⟹ G ⊢ (Normal s) −jvmd→ t ⟹ t ≠ TypeError"
⟨*proof*⟩

corollary *no_type_errors_initial:*
fixes *G (<Γ>)* **and** *Phi (<Φ>)*
assumes *wt: "wt_jvm_prog Γ Φ"*
assumes *is_class: "is_class Γ C"*
and *"method": "method (Γ,C) (m,[]) = Some (C, b)"*
and *m: "m ≠ init"*
defines *start: "s ≡ start_state Γ C m"*

assumes *s: "Γ ⊢ (Normal s) −jvmd→ t"*
shows *"t ≠ TypeError"*
⟨*proof*⟩

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

corollary *welltyped_commutes:*
fixes *G (<Γ>)* **and** *Phi (<Φ>)*
assumes *wt: "wt_jvm_prog Γ Φ"* **and** *∗: "Γ,Φ ⊢JVM s √"*
shows *"Γ ⊢ (Normal s) −jvmd→ (Normal t) = Γ ⊢ s −jvm→ t"*
⟨*proof*⟩

corollary *welltyped_initial_commutes:*
fixes *G (<Γ>)* **and** *Phi (<Φ>)*
assumes *wt: "wt_jvm_prog Γ Φ"*
assumes *is_class: "is_class Γ C"*
and *"method": "method (Γ,C) (m,[]) = Some (C, b)"*
and *m: "m ≠ init"*
defines *start: "s ≡ start_state Γ C m"*
shows *"Γ ⊢ (Normal s) −jvmd→ (Normal t) = Γ ⊢ s −jvm→ t"*
⟨*proof*⟩

end

4.25 Kildall for the JVM

theory *JVM*
imports *Typing_Framework_JVM*
begin

definition *kiljvm :: "jvm_prog ⇒ nat ⇒ nat ⇒ ty ⇒ exception_table ⇒ instr list ⇒ JVMType.state list ⇒ JVMType.state list"* **where**
"kiljvm G maxs maxr rT et bs ==
kildall (JVMType.le G maxs maxr) (JVMType.sup G maxs maxr) (exec G maxs rT et bs)"

definition *wt_kil :: "jvm_prog ⇒ cname ⇒ ty list ⇒ ty ⇒ nat ⇒ nat ⇒ exception_table ⇒ instr list ⇒ bool"* **where**
"wt_kil G C pTs rT mxs mxl et ins ==
check_bounded ins et ∧ 0 < size ins ∧
(let first = Some ([],(OK (Class C))#((map OK pTs))@(replicate mxl Err));
start = OK first#(replicate (size ins - 1) (OK None));
result = kiljvm G mxs (1+size pTs+mxl) rT et ins start
in ∀n < size ins. result!n ≠ Err)"

definition *wt_jvm_prog_kildall :: "jvm_prog ⇒ bool"* **where**
"wt_jvm_prog_kildall G ==
wf_prog (λG C (sig,rT,(maxs,maxl,b,et)). wt_kil G C (snd sig) rT maxs maxl et b) G"

theorem *is_bcv_kiljvm:*
"⟦ wf_prog wf_mb G; bounded (exec G maxs rT et bs) (size bs) ⟧ ⟹
is_bcv (JVMType.le G maxs maxr) Err (exec G maxs rT et bs)
(size bs) (states G maxs maxr) (kiljvm G maxs maxr rT et bs)"
⟨*proof*⟩

lemma *subset_replicate: "set (replicate n x) ⊆ {x}"*
⟨*proof*⟩

lemma *in_set_replicate:*
"x ∈ set (replicate n y) ⟹ x = y"
⟨*proof*⟩

theorem *wt_kil_correct:*
assumes *wf: "wf_prog wf_mb G"*
assumes *C: "is_class G C"*
assumes *pTs: "set pTs ⊆ types G"*

assumes *wtk: "wt_kil G C pTs rT maxs mxl et bs"*

shows *"∃phi. wt_method G C pTs rT maxs mxl bs et phi"*
⟨*proof*⟩

theorem *wt_kil_complete:*
assumes *wf: "wf_prog wf_mb G"*
assumes *C: "is_class G C"*
assumes *pTs: "set pTs ⊆ types G"*

assumes *wtm: "wt_method G C pTs rT maxs mxl bs et phi"*

shows *"wt_kil G C pTs rT maxs mxl et bs"*
⟨*proof*⟩

theorem *jvm_kildall_sound_complete:*

```
  "wt_jvm_prog_kildall G = (∃Phi. wt_jvm_prog G Phi)"
```
⟨*proof*⟩

end

4.26 Example Welltypings

theory *BVExample*
imports
```
  "../JVM/JVMListExample"
  BVSpecTypeSafe
  JVM
```
begin

This theory shows type correctness of the example program in section 3.6 (p. 56) by explicitly providing a welltyping. It also shows that the start state of the program conforms to the welltyping; hence type safe execution is guaranteed.

4.26.1 Setup

Abbreviations for definitions we will have to use often in the proofs below:

```
lemmas name_defs   = list_name_def test_name_def val_name_def next_name_def
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, [],[])"
```
⟨*proof*⟩

lemma *class_NullPointer [simp]:*
```
  "class E (Xcpt NullPointer) = Some (Object, [], [])"
```
⟨*proof*⟩

lemma *class_OutOfMemory [simp]:*
```
  "class E (Xcpt OutOfMemory) = Some (Object, [], [])"
```
⟨*proof*⟩

lemma *class_ClassCast [simp]:*
```
  "class E (Xcpt ClassCast) = Some (Object, [], [])"
```
⟨*proof*⟩

lemma *class_list [simp]:*
```
  "class E list_name = Some list_class"
```
⟨*proof*⟩

lemma *class_test [simp]:*
```
  "class E test_name = Some test_class"
```
⟨*proof*⟩

lemma *E_classes [simp]:*
```
"{C. is_class E C} = {list_name, test_name, Xcpt NullPointer,
                      Xcpt ClassCast, Xcpt OutOfMemory, Object}"
```
⟨*proof*⟩

The subclass releation spelled out:

lemma *subcls1:*
```
"subcls1 E = {(list_name,Object), (test_name,Object), (Xcpt NullPointer, Object),
              (Xcpt ClassCast, Object), (Xcpt OutOfMemory, Object)}"
```
⟨*proof*⟩

The subclass relation is acyclic; hence its converse is well founded:

lemma *notin_rtrancl:*
```
"(a, b) ∈ r* ⟹ a ≠ b ⟹ (⋀y. (a, y) ∉ r) ⟹ False"
```
⟨*proof*⟩

lemma *acyclic_subcls1_E: "acyclic (subcls1 E)"*
⟨*proof*⟩

lemma *wf_subcls1_E: "wf ((subcls1 E)⁻¹)"*
⟨*proof*⟩

Method and field lookup:

lemma *method_Object [simp]:*
```
"method (E, Object) = Map.empty"
```
⟨*proof*⟩

lemma *method_append [simp]:*
```
"method (E, list_name) (append_name, [Class list_name]) =
Some (list_name, PrimT Void, 3, 0, append_ins, [(1, 2, 8, Xcpt NullPointer)])"
```
⟨*proof*⟩

lemma *method_makelist [simp]:*
```
"method (E, test_name) (makelist_name, []) =
Some (test_name, PrimT Void, 3, 2, make_list_ins, [])"
```
⟨*proof*⟩

lemma *field_val [simp]:*
```
"field (E, list_name) val_name = Some (list_name, PrimT Integer)"
```
⟨*proof*⟩

lemma *field_next [simp]:*
```
"field (E, list_name) next_name = Some (list_name, Class list_name)"
```
⟨*proof*⟩

lemma *[simp]: "fields (E, Object) = []"*
⟨*proof*⟩

lemma *[simp]: "fields (E, Xcpt NullPointer) = []"*
⟨*proof*⟩

lemma *[simp]: "fields (E, Xcpt ClassCast) = []"*
⟨*proof*⟩

lemma *[simp]: "fields (E, Xcpt OutOfMemory) = []"*
⟨*proof*⟩

lemma *[simp]: "fields (E, test_name) = []"*
⟨*proof*⟩

lemmas *[simp] = is_class_def*

The next definition and three proof rules implement an algorithm to enumarate natural numbers. The command *apply (elim pc_end pc_next pc_0* transforms a goal of the form

pc < n ⟹ P pc

into a series of goals

P 0

P (Suc 0)

...

P n

definition *intervall :: "nat ⇒ nat ⇒ nat ⇒ bool" (‹_ ∈ [_, _')›)* **where**
"x ∈ [a, b) ≡ a ≤ x ∧ x < b"

lemma *pc_0: "x < n ⟹ (x ∈ [0, n) ⟹ P x) ⟹ P x"*
⟨*proof*⟩

lemma *pc_next: "x ∈ [n0, n) ⟹ P n0 ⟹ (x ∈ [Suc n0, n) ⟹ P x) ⟹ P x"*
⟨*proof*⟩

lemma *pc_end: "x ∈ [n,n) ⟹ P x"*
⟨*proof*⟩

4.26.2 Program structure

The program is structurally wellformed:

lemma *wf_struct:*
"wf_prog (λG C mb. True) E" (**is** *"wf_prog ?mb E"*)
⟨*proof*⟩

4.26.3 Welltypings

We show welltypings of the methods *append_name* in class *list_name*, and *makelist_name* in class *test_name*:

lemmas *eff_simps [simp] = eff_def norm_eff_def xcpt_eff_def*
declare *appInvoke [simp del]*

definition *phi_append :: method_type (‹φa›)* **where**
"φa ≡ 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]),
  ([NT, 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]),
  (                          [PrimT Void], [Class list_name, Class list_name]),
  (                       [Class Object], [Class list_name, Class list_name]),
  (                                  [], [Class list_name, Class list_name]),
  (                     [Class list_name], [Class list_name, Class list_name]),
  (    [Class list_name, Class list_name], [Class list_name, Class list_name]),
  (                                  [], [Class list_name, Class list_name]),
  (                          [PrimT Void], [Class list_name, Class list_name])]"
```

lemma *bounded_append [simp]:*
```
  "check_bounded append_ins [(Suc 0, 2, 8, Xcpt NullPointer)]"
```
⟨*proof*⟩

lemma *types_append [simp]: "check_types E 3 (Suc (Suc 0)) (map OK* φ_a*)"*
⟨*proof*⟩

lemma *wt_append [simp]:*
```
  "wt_method E list_name [Class list_name] (PrimT Void) 3 0 append_ins
            [(Suc 0, 2, 8, Xcpt NullPointer)] φa"
```
⟨*proof*⟩

Some abbreviations for readability

abbreviation *Clist :: ty*
  where *"Clist == Class list_name"*
abbreviation *Ctest :: ty*
  where *"Ctest == Class test_name"*

definition *phi_makelist :: method_type (‹*φ_m*›)* **where**
```
  "φm ≡ map (λ(x,y). Some (x, y)) [
    (                                  [], [OK Ctest, Err     , Err     ]),
    (                               [Clist], [OK Ctest, Err     , Err     ]),
    (                        [Clist, Clist], [OK Ctest, Err     , Err     ]),
    (                               [Clist], [OK Clist, Err     , Err     ]),
    (                 [PrimT Integer, Clist], [OK Clist, Err     , Err     ]),
    (                                  [], [OK Clist, Err     , Err     ]),
    (                               [Clist], [OK Clist, Err     , Err     ]),
    (                        [Clist, Clist], [OK Clist, Err     , Err     ]),
    (                               [Clist], [OK Clist, OK Clist, Err     ]),
    (                 [PrimT Integer, Clist], [OK Clist, OK Clist, Err     ]),
    (                                  [], [OK Clist, OK Clist, Err     ]),
    (                               [Clist], [OK Clist, OK Clist, Err     ]),
    (                        [Clist, Clist], [OK Clist, OK Clist, Err     ]),
    (                               [Clist], [OK Clist, OK Clist, OK Clist]),
    (                 [PrimT Integer, Clist], [OK Clist, OK Clist, OK Clist]),
    (                                  [], [OK Clist, OK Clist, OK Clist]),
    (                               [Clist], [OK Clist, OK Clist, OK Clist]),
    (                        [Clist, Clist], [OK Clist, OK Clist, OK Clist]),
```

```
    (                        [PrimT Void], [OK Clist, OK Clist, OK Clist]),
    (                                  [], [OK Clist, OK Clist, OK Clist]),
    (                             [Clist], [OK Clist, OK Clist, OK Clist]),
    (                       [Clist, Clist], [OK Clist, OK Clist, OK Clist]),
    (                        [PrimT Void], [OK Clist, OK Clist, OK Clist])]"
```

lemma *bounded_makelist [simp]: "check_bounded make_list_ins []"*
⟨*proof*⟩

lemma *types_makelist [simp]: "check_types E 3 (Suc (Suc (Suc 0))) (map OK* φ_m*)"*
⟨*proof*⟩

lemma *wt_makelist [simp]:*
"wt_method E test_name [] (PrimT Void) 3 2 make_list_ins [] φ_m*"*
⟨*proof*⟩

The whole program is welltyped:

definition *Phi :: prog_type (‹Φ›)* **where**
"Φ C sg ≡ if C = test_name ∧ sg = (makelist_name, []) then φ_m *else*
if C = list_name ∧ sg = (append_name, [Class list_name]) then φ_a *else []"*

lemma *wf_prog:*
"wt_jvm_prog E Φ"
⟨*proof*⟩

4.26.4 Conformance

Execution of the program will be typesafe, because its start state conforms to the welltyping:

lemma *"E,Φ ⊢JVM start_state E test_name makelist_name √"*
⟨*proof*⟩

4.26.5 Example for code generation: inferring method types

definition *test_kil :: "jvm_prog ⇒ cname ⇒ ty list ⇒ ty ⇒ nat ⇒ nat ⇒*
exception_table ⇒ instr list ⇒ JVMType.state list" **where**
"test_kil G C pTs rT mxs mxl et instr =
(let first = Some ([],(OK (Class C))#((map OK pTs))@(replicate mxl Err));
start = OK first#(replicate (size instr - 1) (OK None))
in kiljvm G mxs (1+size pTs+mxl) rT et instr start)"

lemma *[code]:*
"unstables r step ss =
fold (λp A. if ¬stable r step ss p then insert p A else A) [0..<size ss] {}"
⟨*proof*⟩

definition *some_elem :: "'a set ⇒ 'a"*
where *[code drop]: "some_elem = (λS. 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!

lemma *False*
⟨*proof*⟩

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)"
```
⟨*proof*⟩

lemma *JVM_sup_unfold [code]:*
```
"JVMType.sup S m n = lift2 (Opt.sup
      (Product.sup (Listn.sup (JType.sup S))
        (λx y. OK (map2 (lift2 (JType.sup S)) x y))))"
```
⟨*proof*⟩

lemmas *[code] =*
```
JType.sup_def [unfolded exec_lub_def]
JVM_le_unfold
lesub_def
plussub_def
wf_class_code
widen.equation
match_exception_entry_def
```

definition *test1* **where**
```
"test1 = test_kil E list_name [Class list_name] (PrimT Void) 3 0
  [(Suc 0, 2, 8, Xcpt NullPointer)] append_ins"
```
definition *test2* **where**
```
"test2 = test_kil E test_name [] (PrimT Void) 3 2 [] make_list_ins"
```

⟨*ML*⟩

end

theory *AuxLemmas*
imports *"../J/JBasis"*
begin

lemma *app_nth_greater_len [simp]:*
```
"length pre ≤ ind ⟹ (pre @ a # post) ! (Suc ind) = (pre @ post) ! ind"
```
⟨*proof*⟩

lemma *length_takeWhile: "v ∈ set xs ⟹ length (takeWhile (λz. z ≠ v) xs) < length xs"*
⟨*proof*⟩

lemma *nth_length_takeWhile [simp]:*
"v ∈ set xs ⟹ xs ! (length (takeWhile (%z. z~=v) xs)) = v"
⟨*proof*⟩

lemma *map_list_update [simp]:*
"⟦ x ∈ set xs; distinct xs⟧ ⟹
(map f xs) [length (takeWhile (λz. z ≠ x) xs) := v] = map (f(x:=v)) xs"
⟨*proof*⟩

lemma *split_compose:*
"(case_prod f) ∘ (λ (a,b). ((fa a), (fb b))) = (λ (a,b). (f (fa a) (fb b)))"
⟨*proof*⟩

lemma *split_iter:*
"(λ (a,b,c). ((g1 a), (g2 b), (g3 c))) = (λ (a,p). ((g1 a), (λ (b, c). ((g2 b), (g3 c))) p))"
⟨*proof*⟩

lemma *singleton_in_set: "A = {a} ⟹ a ∈ A"* ⟨*proof*⟩

lemma *the_map_upd: "(the ∘ f(x↦v)) = (the ∘ f)(x:=v)"*
⟨*proof*⟩

lemma *map_of_in_set:*
"(map_of xs x = None) = (x ∉ set (map fst xs))"
⟨*proof*⟩

lemma *map_map_upd [simp]:*
"y ∉ set xs ⟹ map (the ∘ f(y↦v)) xs = map (the ∘ f) xs"
⟨*proof*⟩

lemma *map_map_upds [simp]:*
"(∀y∈set ys. y ∉ set xs) ⟹ map (the ∘ f(ys[↦]vs)) xs = map (the ∘ f) xs"
⟨*proof*⟩

lemma *map_upds_distinct [simp]:*
"distinct ys ⟹ length ys = length vs ⟹ map (the ∘ f(ys[↦]vs)) ys = vs"

⟨*proof*⟩

lemma *map_of_map_as_map_upd:*
"distinct (map f zs) ⟹ map_of (map (λ p. (f p, g p)) zs) = Map.empty (map f zs [↦] map g zs)"
⟨*proof*⟩

lemma *map_upds_SomeD:*
"(m(xs[↦]ys)) k = Some y ⟹ k ∈ (set xs) ∨ (m k = Some y)"
⟨*proof*⟩

lemma *map_of_upds_SomeD: "((map_of m) (xs[↦]ys)) k = Some y*
⟹ k ∈ (set (xs @ map fst m))"
⟨*proof*⟩

lemma *map_of_map_prop:*
"⟦map_of (map f xs) k = Some v; ∀x ∈ set xs. P1 x; ∀x. P1 x ⟶ P2 (f x)⟧ ⟹ P2 (k, v)"
⟨*proof*⟩

lemma *map_of_map2: "∀x ∈ set xs. (fst (f x)) = (fst x) ⟹*
map_of (map f xs) a = map_option (λ b. (snd (f (a, b)))) (map_of xs a)"
⟨*proof*⟩

end

theory *DefsComp*
imports *"../JVM/JVMExec"*
begin

definition *method_rT :: "cname × ty × 'c ⇒ ty"* **where**
"method_rT mtd == (fst (snd mtd))"

definition
gx :: "xstate ⇒ val option" **where** *"gx ≡ fst"*

definition
gs :: "xstate ⇒ state" **where** *"gs ≡ snd"*

definition
gh :: "xstate ⇒ aheap" **where** *"gh ≡ fst∘snd"*

definition
gl :: "xstate ⇒ State.locals" **where** *"gl ≡ snd∘snd"*

definition
gmb :: "'a prog ⇒ cname ⇒ sig ⇒ 'a"

```
    where "gmb G cn si ≡ snd(snd(the(method (G,cn) si)))"

definition
  gis :: "jvm_method ⇒ bytecode"
    where "gis ≡ fst ∘ snd ∘ snd"


definition
  gjmb_pns  :: "java_mb ⇒ vname list" where "gjmb_pns ≡ fst"

definition
  gjmb_lvs  :: "java_mb ⇒ (vname×ty)list" where  "gjmb_lvs ≡ fst∘snd"

definition
  gjmb_blk  :: "java_mb ⇒ stmt" where  "gjmb_blk ≡ fst∘snd∘snd"

definition
  gjmb_res  :: "java_mb ⇒ expr" where  "gjmb_res ≡ snd∘snd∘snd"

definition
  gjmb_plns :: "java_mb ⇒ vname list"
    where  "gjmb_plns ≡ λjmb. gjmb_pns jmb @ map fst (gjmb_lvs jmb)"

definition
  glvs :: "java_mb ⇒ State.locals ⇒ locvars"
    where "glvs jmb loc ≡ map (the∘loc) (gjmb_plns jmb)"

lemmas gdefs = gx_def gh_def gl_def gmb_def gis_def glvs_def
lemmas gjmbdefs = gjmb_pns_def gjmb_lvs_def gjmb_blk_def gjmb_res_def gjmb_plns_def

lemmas galldefs = gdefs gjmbdefs

definition locvars_locals :: "java_mb prog ⇒ cname ⇒ sig ⇒ State.locals ⇒ locvars"
where
  "locvars_locals G C S lvs == the (lvs This) # glvs (gmb G C S) lvs"

definition locals_locvars :: "java_mb prog ⇒ cname ⇒ sig ⇒ locvars ⇒ State.locals"
where
  "locals_locvars G C S lvs ==
  Map.empty ((gjmb_plns (gmb G C S))[↦](tl lvs), This↦(hd lvs))"

definition locvars_xstate :: "java_mb prog ⇒ cname ⇒ sig ⇒ xstate ⇒ locvars" where
  "locvars_xstate G C S xs == locvars_locals G C S (gl xs)"


lemma locvars_xstate_par_dep:
  "lv1 = lv2 ⟹
  locvars_xstate G C S (xcpt1, hp1, lv1) = locvars_xstate G C S (xcpt2, hp2, lv2)"
⟨proof⟩
```

lemma *gx_conv [simp]: "gx (xcpt, s) = xcpt"* ⟨*proof*⟩

lemma *gh_conv [simp]: "gh (xcpt, h, l) = h"* ⟨*proof*⟩

end

theory *Index*
imports *AuxLemmas DefsComp*
begin

definition *index :: "java_mb => vname => nat"* **where**
"index == λ (pn,lv,blk,res) v.
if v = This
then 0
else Suc (length (takeWhile (λ z. z~=v) (pn @ map fst lv)))"

lemma *index_length_pns: "*
⟦ *i = index (pns,lvars,blk,res) vn;*
wf_java_mdecl G C ((mn,pTs),rT, (pns,lvars,blk,res));
vn ∈ set pns⟧
⟹ 0 < i ∧ i < Suc (length pns)"
⟨*proof*⟩

lemma *index_length_lvars: "*
⟦ *i = index (pns,lvars,blk,res) vn;*
wf_java_mdecl G C ((mn,pTs),rT, (pns,lvars,blk,res));
vn ∈ set (map fst lvars)⟧
⟹ (length pns) < i ∧ i < Suc((length pns) + (length lvars))"
⟨*proof*⟩

lemma *select_at_index :*
"x ∈ set (gjmb_plns (gmb G C S)) ∨ x = This
⟹ (the (loc This) # glvs (gmb G C S) loc) ! (index (gmb G C S) x) = the (loc x)"
⟨*proof*⟩

lemma *lift_if: "(f (if b then t else e)) = (if b then (f t) else (f e))"*
⟨*proof*⟩

lemma *update_at_index: "*
⟦ *distinct (gjmb_plns (gmb G C S));*
x ∈ set (gjmb_plns (gmb G C S)); x ≠ This ⟧ *⟹*
(locvars_xstate G C S (Norm (h, l)))[index (gmb G C S) x := val] =

```
  locvars_xstate G C S (Norm (h, l(x↦val)))"
```
⟨*proof*⟩

lemma *index_of_var:* "⟦ *xvar ∉ set pns; xvar ∉ set (map fst zs); xvar ≠ This* ⟧
⟹ *index (pns, zs @ ((xvar, xval) # xys), blk, res) xvar = Suc (length pns + length zs)*"
⟨*proof*⟩

definition *disjoint_varnames :: "[vname list, (vname × ty) list] ⇒ bool"* **where**
"disjoint_varnames pns lvars ≡
distinct pns ∧ unique lvars ∧ This ∉ set pns ∧ This ∉ set (map fst lvars) ∧
(∀ pn∈set pns. pn ∉ set (map fst lvars))"

lemma *index_of_var2: "*
disjoint_varnames pns (lvars_pre @ (vn, ty) # lvars_post)
⟹ *index (pns, lvars_pre @ (vn, ty) # lvars_post, blk, res) vn =*
Suc (length pns + length lvars_pre)"
⟨*proof*⟩

lemma *wf_java_mdecl_disjoint_varnames:*
"wf_java_mdecl G C (S,rT,(pns,lvars,blk,res))
⟹ *disjoint_varnames pns lvars"*
⟨*proof*⟩

lemma *wf_java_mdecl_length_pTs_pns:*
"wf_java_mdecl G C ((mn, pTs), rT, pns, lvars, blk, res)
⟹ *length pTs = length pns"*
⟨*proof*⟩

end

theory *TranslCompTp*
imports *Index "../BV/JVMType"*
begin

definition *comb :: "['a ⇒ 'b list × 'c, 'c ⇒ 'b list × 'd, 'a] ⇒ 'b list × 'd"*
*(***infixr** *‹□› 55)*
where
```
  "comb == (λ f1 f2 x0. let (xs1, x1) = f1 x0;
                            (xs2, x2) = f2 x1
                        in  (xs1 @ xs2, x2))"
```

definition *comb_nil :: "'a ⇒ 'b list × 'a"* **where**
"comb_nil a == ([], a)"

lemma *comb_nil_left [simp]: "comb_nil □ f = f"*
⟨*proof*⟩

lemma *comb_nil_right [simp]: "f □ comb_nil = f"*
⟨*proof*⟩

lemma *comb_assoc [simp]: "(fa □ fb) □ fc = fa □ (fb □ fc)"*
⟨*proof*⟩

lemma *comb_inv:*
"(xs', x') = (f1 □ f2) x0 ⟹
∃xs1 x1 xs2 x2. (xs1, x1) = (f1 x0) ∧ (xs2, x2) = f2 x1 ∧ xs'= xs1 @ xs2 ∧ x'=x2"
⟨*proof*⟩

abbreviation *(input)*
mt_of :: "method_type × state_type ⇒ method_type"
where *"mt_of == fst"*

abbreviation *(input)*
sttp_of :: "method_type × state_type ⇒ state_type"
where *"sttp_of == snd"*

definition *nochangeST :: "state_type ⇒ method_type × state_type"* **where**
"nochangeST sttp == ([Some sttp], sttp)"

definition *pushST :: "[ty list, state_type] ⇒ method_type × state_type"* **where**
"pushST tps == (λ (ST, LT). ([Some (ST, LT)], (tps @ ST, LT)))"

definition *dupST :: "state_type ⇒ method_type × state_type"* **where**
"dupST == (λ (ST, LT). ([Some (ST, LT)], (hd ST # ST, LT)))"

definition *dup_x1ST :: "state_type ⇒ method_type × state_type"* **where**
"dup_x1ST == (λ (ST, LT). ([Some (ST, LT)],
(hd ST # hd (tl ST) # hd ST # (tl (tl ST)), LT)))"

definition *popST :: "[nat, state_type] ⇒ method_type × state_type"* **where**
"popST n == (λ (ST, LT). ([Some (ST, LT)], (drop n ST, LT)))"

definition *replST :: "[nat, ty, state_type] ⇒ method_type × state_type"* **where**
"replST n tp == (λ (ST, LT). ([Some (ST, LT)], (tp # (drop n ST), LT)))"

definition *storeST :: "[nat, ty, state_type] ⇒ method_type × state_type"* **where**
"storeST i tp == (λ (ST, LT). ([Some (ST, LT)], (tl ST, LT [i:= OK tp])))"

primrec *compTpExpr :: "java_mb ⇒ java_mb prog ⇒ expr ⇒*
state_type ⇒ method_type × state_type"
and *compTpExprs :: "java_mb ⇒ java_mb prog ⇒ expr list ⇒*
state_type ⇒ method_type × state_type"

```
where
  "compTpExpr jmb G (NewC c) = pushST [Class c]"
| "compTpExpr jmb G (Cast c e) = (compTpExpr jmb G e) □ (replST 1 (Class c))"
| "compTpExpr jmb G (Lit val) = pushST [the (typeof (λv. None) val)]"
| "compTpExpr jmb G (BinOp bo e1 e2) =
     (compTpExpr jmb G e1) □ (compTpExpr jmb G e2) □
     (case bo of
       Eq => popST 2 □ pushST [PrimT Boolean] □ popST 1 □ pushST [PrimT Boolean]
     | Add => replST 2 (PrimT Integer))"
| "compTpExpr jmb G (LAcc vn) = (λ (ST, LT).
      pushST [ok_val (LT ! (index jmb vn))] (ST, LT))"
| "compTpExpr jmb G (vn::=e) =
      (compTpExpr jmb G e) □ dupST □ (popST 1)"
| "compTpExpr jmb G ( {cn}e..fn ) =
      (compTpExpr jmb G e) □ replST 1 (snd (the (field (G,cn) fn)))"
| "compTpExpr jmb G (FAss cn e1 fn e2 ) =
      (compTpExpr jmb G e1) □ (compTpExpr jmb G e2) □ dup_x1ST □ (popST 2)"
| "compTpExpr jmb G ({C}a..mn({fpTs}ps)) =
       (compTpExpr jmb G a) □ (compTpExprs jmb G ps) □
       (replST ((length ps) + 1) (method_rT (the (method (G,C) (mn,fpTs)))))"
| "compTpExprs jmb G [] = comb_nil"
| "compTpExprs jmb G (e#es) = (compTpExpr jmb G e) □ (compTpExprs jmb G es)"



primrec compTpStmt :: "java_mb ⇒ java_mb prog ⇒ stmt ⇒
    state_type ⇒ method_type × state_type"
where
  "compTpStmt jmb G Skip = comb_nil"
| "compTpStmt jmb G (Expr e) =  (compTpExpr jmb G e) □ popST 1"
| "compTpStmt jmb G (c1;; c2) = (compTpStmt jmb G c1) □ (compTpStmt jmb G c2)"
| "compTpStmt jmb G (If(e) c1 Else c2) =
      (pushST [PrimT Boolean]) □ (compTpExpr jmb G e) □ popST 2 □
      (compTpStmt jmb G c1) □ nochangeST □ (compTpStmt jmb G c2)"
| "compTpStmt jmb G (While(e) c) =
      (pushST [PrimT Boolean]) □ (compTpExpr jmb G e) □ popST 2 □
      (compTpStmt jmb G c) □ nochangeST"

definition compTpInit :: "java_mb ⇒ (vname * ty)
                  ⇒ state_type ⇒ method_type × state_type" where
  "compTpInit jmb == (λ (vn,ty). (pushST [ty]) □  (storeST (index jmb vn) ty))"

primrec compTpInitLvars :: "[java_mb, (vname × ty) list] ⇒
    state_type ⇒ method_type × state_type"
where
  "compTpInitLvars jmb [] = comb_nil"
| "compTpInitLvars jmb (lv#lvars) = (compTpInit jmb lv) □ (compTpInitLvars jmb lvars)"

definition start_ST :: "opstack_type" where
  "start_ST == []"

definition start_LT :: "cname ⇒ ty list ⇒ nat ⇒ locvars_type" where
  "start_LT C pTs n ==  (OK (Class C))#((map OK pTs))@(replicate n Err)"
```

definition *compTpMethod :: "[java_mb prog, cname, java_mb mdecl] ⇒ method_type"* **where**
"compTpMethod G C == λ ((mn,pTs),rT, jmb).
let (pns,lvars,blk,res) = jmb
in (mt_of
((compTpInitLvars jmb lvars □
compTpStmt jmb G blk □
compTpExpr jmb G res □
nochangeST)
(start_ST, start_LT C pTs (length lvars))))"

definition *compTp :: "java_mb prog => prog_type"* **where**
"compTp G C sig == let (D, rT, jmb) = (the (method (G, C) sig))
in compTpMethod G C (sig, rT, jmb)"

definition *ssize_sto :: "(state_type option) ⇒ nat"* **where**
"ssize_sto sto == case sto of None ⇒ 0 | (Some (ST, LT)) ⇒ length ST"

definition *max_of_list :: "nat list ⇒ nat"* **where**
"max_of_list xs == foldr max xs 0"

definition *max_ssize :: "method_type ⇒ nat"* **where**
"max_ssize mt == max_of_list (map ssize_sto mt)"

end

theory *TranslComp* **imports** *TranslCompTp* **begin**

primrec *compExpr :: "java_mb => expr => instr list"*
and *compExprs :: "java_mb => expr list => instr list"*
where

"compExpr jmb (NewC c) = [New c]" |

"compExpr jmb (Cast c e) = compExpr jmb e @ [Checkcast c]" |

```
"compExpr jmb (Lit val) = [LitPush val]" |


"compExpr jmb (BinOp bo e1 e2) = compExpr jmb e1 @ compExpr jmb e2 @
  (case bo of Eq => [Ifcmpeq 3,LitPush(Bool False),Goto 2,LitPush(Bool True)]
            | Add => [IAdd])" |


"compExpr jmb (LAcc vn) = [Load (index jmb vn)]" |


"compExpr jmb (vn::=e) = compExpr jmb e @ [Dup , Store (index jmb vn)]" |


"compExpr jmb ( {cn}e..fn ) = compExpr jmb e @ [Getfield fn cn]" |


"compExpr jmb (FAss cn e1 fn e2 ) =
       compExpr jmb e1 @ compExpr jmb e2 @ [Dup_x1 , Putfield fn cn]" |


"compExpr jmb (Call cn e1 mn X ps) =
        compExpr jmb e1 @ compExprs jmb ps @ [Invoke cn mn X]" |



"compExprs jmb []     = []" |

"compExprs jmb (e#es) = compExpr jmb e @ compExprs jmb es"


primrec compStmt :: "java_mb => stmt => instr list" where


"compStmt jmb Skip = []" |

"compStmt jmb (Expr e) = ((compExpr jmb e) @ [Pop])" |

"compStmt jmb (c1;; c2) = ((compStmt jmb c1) @ (compStmt jmb c2))" |

"compStmt jmb (If(e) c1 Else c2) =
        (let cnstf = LitPush (Bool False);
             cnd   = compExpr jmb e;
             thn   = compStmt jmb c1;
             els   = compStmt jmb c2;
             test  = Ifcmpeq (int(length thn +2));
             thnex = Goto (int(length els +1))
```

```
         in
         [cnstf] @ cnd @ [test] @ thn @ [thnex] @ els)" |

"compStmt jmb (While(e) c) =
        (let cnstf = LitPush (Bool False);
             cnd   = compExpr jmb e;
             bdy   = compStmt jmb c;
             test  = Ifcmpeq (int(length bdy +2));
             loop  = Goto (-(int((length bdy) + (length cnd) +2)))
         in
         [cnstf] @ cnd @ [test] @ bdy @ [loop])"
```

definition *load_default_val :: "ty => instr"* **where**
"load_default_val ty == LitPush (default_val ty)"

definition *compInit :: "java_mb => (vname * ty) => instr list"* **where**
"compInit jmb == λ (vn,ty). [load_default_val ty, Store (index jmb vn)]"

definition *compInitLvars :: "[java_mb, (vname × ty) list] ⇒ bytecode"* **where**
"compInitLvars jmb lvars == concat (map (compInit jmb) lvars)"

definition *compMethod :: "java_mb prog ⇒ cname ⇒ java_mb mdecl ⇒ jvm_method mdecl"* **where**

```
  "compMethod G C jmdl == let (sig, rT, jmb) = jmdl;
                          (pns,lvars,blk,res) = jmb;
                          mt = (compTpMethod G C jmdl);
                          bc = compInitLvars jmb lvars @
                               compStmt jmb blk @ compExpr jmb res @
                               [Return]
                  in (sig, rT, max_ssize mt, length lvars, bc, [])"
```

definition *compClass :: "java_mb prog => java_mb cdecl=> jvm_method cdecl"* **where**
"compClass G == λ (C,cno,fdls,jmdls). (C,cno,fdls, map (compMethod G C) jmdls)"

definition *comp :: "java_mb prog => jvm_prog"* **where**
"comp G == map (compClass G) G"

end

theory *LemmasComp*
imports *TranslComp*
begin

context
begin

declare *split_paired_All [simp del]*
declare *split_paired_Ex [simp del]*

lemma *c_hupd_conv:*
"c_hupd h' (xo, (h,l)) = (xo, (if xo = None then h' else h),l)"
⟨*proof*⟩

lemma *gl_c_hupd [simp]: "(gl (c_hupd h xs)) = (gl xs)"*
⟨*proof*⟩

lemma *c_hupd_xcpt_invariant [simp]: "gx (c_hupd h' (xo, st)) = xo"*
⟨*proof*⟩

lemma *c_hupd_hp_invariant: "gh (c_hupd hp (None, st)) = hp"*
⟨*proof*⟩

lemma *unique_map_fst [rule_format]: "(∀ x ∈ set xs. (fst x = fst (f x))) ⟶*
unique (map f xs) = unique xs"
⟨*proof*⟩

lemma *comp_unique: "unique (comp G) = unique G"*
⟨*proof*⟩

lemma *comp_class_imp:*
"(class G C = Some(D, fs, ms)) ⟹
(class (comp G) C = Some(D, fs, map (compMethod G C) ms))"
⟨*proof*⟩

lemma *comp_class_None:*
"(class G C = None) = (class (comp G) C = None)"
⟨*proof*⟩

lemma *comp_is_class: "is_class (comp G) C = is_class G C"*
⟨*proof*⟩

lemma *comp_is_type: "is_type (comp G) T = is_type G T"*
⟨*proof*⟩

lemma *comp_classname:*
"is_class G C ⟹ fst (the (class G C)) = fst (the (class (comp G) C))"
⟨*proof*⟩

lemma *comp_subcls1: "subcls1 (comp G) = subcls1 G"*

⟨*proof*⟩

lemma *comp_widen: "widen (comp G) = widen G"*
⟨*proof*⟩

lemma *comp_cast: "cast (comp G) = cast G"*
⟨*proof*⟩

lemma *comp_cast_ok: "cast_ok (comp G) = cast_ok G"*
⟨*proof*⟩

lemma *compClass_fst [simp]: "(fst (compClass G C)) = (fst C)"*
⟨*proof*⟩

lemma *compClass_fst_snd [simp]: "(fst (snd (compClass G C))) = (fst (snd C))"*
⟨*proof*⟩

lemma *compClass_fst_snd_snd [simp]: "(fst (snd (snd (compClass G C)))) = (fst (snd (snd C)))"*
⟨*proof*⟩

lemma *comp_wf_fdecl [simp]: "wf_fdecl (comp G) fd = wf_fdecl G fd"*
⟨*proof*⟩

lemma *compClass_forall [simp]:*
"(∀x∈set (snd (snd (snd (compClass G C)))). P (fst x) (fst (snd x))) =
(∀x∈set (snd (snd (snd C))). P (fst x) (fst (snd x)))"
⟨*proof*⟩

lemma *comp_wf_mhead: "wf_mhead (comp G) S rT = wf_mhead G S rT"*
⟨*proof*⟩

lemma *comp_ws_cdecl:*
"ws_cdecl (TranslComp.comp G) (compClass G C) = ws_cdecl G C"
⟨*proof*⟩

lemma *comp_wf_syscls: "wf_syscls (comp G) = wf_syscls G"*
⟨*proof*⟩

lemma *comp_ws_prog: "ws_prog (comp G) = ws_prog G"*
⟨*proof*⟩

lemma *comp_class_rec:*
"wf ((subcls1 G)$^{-1}$) ⟹
class_rec (comp G) C t f =
class_rec G C t (λ C' fs' ms' r'. f C' fs' (map (compMethod G C') ms') r')"
⟨*proof*⟩

lemma *comp_fields: "wf ((subcls1 G)$^{-1}$) ⟹*

```
  fields (comp G,C) = fields (G,C)"
```
⟨*proof*⟩

lemma *comp_field: "wf ((subcls1 G)⁻¹) ⟹*
```
  field (comp G,C) = field (G,C)"
```
⟨*proof*⟩

lemma *class_rec_relation [rule_format (no_asm)]: "⟦ ws_prog G;*
```
  ∀fs ms. R (f1 Object fs ms t1) (f2 Object fs ms t2);
   ∀C fs ms r1 r2. (R r1 r2) ⟶ (R (f1 C fs ms r1) (f2 C fs ms r2)) ⟧
  ⟹ ((class G C) ≠ None) ⟶  R (class_rec G C t1 f1) (class_rec G C t2 f2)"
```
⟨*proof*⟩

abbreviation *(input)*
```
  "mtd_mb == snd o snd"
```

lemma *map_of_map:*
```
  "map_of (map (λ(k, v). (k, f v)) xs) k = map_option f (map_of xs k)"
```
⟨*proof*⟩

lemma *map_of_map_fst:*
```
  "⟦ inj f; ∀x∈set xs. fst (f x) = fst x; ∀x∈set xs. fst (g x) = fst x ⟧
  ⟹  map_of (map g xs) k = map_option (λ e. (snd (g ((inv f) (k, e))))) (map_of (map
f xs) k)"
```
⟨*proof*⟩

lemma *comp_method [rule_format (no_asm)]:*
```
  "⟦ ws_prog G; is_class G C⟧ ⟹
  ((method (comp G, C) S) =
   map_option (λ (D,rT,b).  (D, rT, mtd_mb (compMethod G D (S, rT, b))))
              (method (G, C) S))"
```
⟨*proof*⟩

lemma *comp_wf_mrT: "⟦ ws_prog G; is_class G D⟧ ⟹*
```
  wf_mrT (TranslComp.comp G) (C, D, fs, map (compMethod G a) ms) =
  wf_mrT G (C, D, fs, ms)"
```
⟨*proof*⟩

lemma *max_spec_preserves_length:*
```
  "max_spec G C (mn, pTs) = {((md,rT),pTs')} ⟹ length pTs = length pTs'"
```
⟨*proof*⟩

lemma *ty_exprs_length [simp]: "(E⊢es[::]Ts ⟶ length es = length Ts)"*
⟨*proof*⟩

lemma *max_spec_preserves_method_rT [simp]:*
"max_spec G C (mn, pTs) = {((md,rT),pTs')}
⟹ method_rT (the (method (G, C) (mn, pTs'))) = rT"
⟨*proof*⟩

end

declare *compClass_fst [simp del]*
declare *compClass_fst_snd [simp del]*
declare *compClass_fst_snd_snd [simp del]*

end

theory *CorrComp*
imports *"../J/JTypeSafe" LemmasComp*
begin

declare *wf_prog_ws_prog [simp add]*

lemma *eval_evals_exec_xcpt:*
"(G ⊢ xs -ex≻val-> xs' ⟶ gx xs' = None ⟶ gx xs = None) ∧
(G ⊢ xs -exs[≻]vals-> xs' ⟶ gx xs' = None ⟶ gx xs = None) ∧
(G ⊢ xs -st-> xs' ⟶ gx xs' = None ⟶ gx xs = None)"
⟨*proof*⟩

lemma *eval_xcpt: "G ⊢ xs -ex≻val-> xs' ⟹ gx xs' = None ⟹ gx xs = None"*
(is *"?H1 ⟹ ?H2 ⟹ ?T")*
⟨*proof*⟩

lemma *evals_xcpt: "G ⊢ xs -exs[≻]vals-> xs' ⟹ gx xs' = None ⟹ gx xs = None"*
(is *"?H1 ⟹ ?H2 ⟹ ?T")*
⟨*proof*⟩

lemma *exec_xcpt: "G ⊢ xs -st-> xs' ⟹ gx xs' = None ⟹ gx xs = None"*
(is *"?H1 ⟹ ?H2 ⟹ ?T")*
⟨*proof*⟩

theorem *exec_all_trans: "⟦(exec_all G s0 s1); (exec_all G s1 s2)⟧ ⟹ (exec_all G s0 s2)"*
⟨*proof*⟩

theorem *exec_all_refl: "exec_all G s s"*
⟨*proof*⟩

theorem *exec_instr_in_exec_all:*
"⟦ exec_instr i G hp stk lvars C S pc frs = (None, hp', frs');
gis (gmb G C S) ! pc = i⟧ ⟹
G ⊢ (None, hp, (stk, lvars, C, S, pc) # frs) −jvm→ (None, hp', frs')"
⟨*proof*⟩

theorem *exec_all_one_step: "*
⟦ gis (gmb G C S) = pre @ (i # post); pc0 = length pre;
(exec_instr i G hp0 stk0 lvars0 C S pc0 frs) =
(None, hp1, (stk1,lvars1,C,S, Suc pc0)#frs) ⟧
⟹
G ⊢ (None, hp0, (stk0,lvars0,C,S, pc0)#frs) −jvm→
(None, hp1, (stk1,lvars1,C,S, Suc pc0)#frs)"
⟨*proof*⟩

definition *progression :: "jvm_prog ⇒ cname ⇒ sig ⇒*
aheap ⇒ opstack ⇒ locvars ⇒
bytecode ⇒
aheap ⇒ opstack ⇒ locvars ⇒
bool"
(‹{_,_,_} ⊢ {_, _, _} >- _ → {_, _, _}› [61,61,61,61,61,61,90,61,61,61]60) **where**
"{G,C,S} ⊢ {hp0, os0, lvars0} >- instrs → {hp1, os1, lvars1} ==
∀ pre post frs.
(gis (gmb G C S) = pre @ instrs @ post) ⟶
G ⊢ (None,hp0,(os0,lvars0,C,S,length pre)#frs) −jvm→
(None,hp1,(os1,lvars1,C,S,(length pre) + (length instrs))#frs)"

lemma *progression_call:*
"⟦ ∀ pc frs.
exec_instr instr G hp0 os0 lvars0 C S pc frs =
(None, hp', (os', lvars', C', S', 0) # (fr pc) # frs) ∧
gis (gmb G C' S') = instrs' @ [Return] ∧
{G, C', S'} ⊢ {hp', os', lvars'} >- instrs' → {hp'', os'', lvars''} ∧
exec_instr Return G hp'' os'' lvars'' C' S' (length instrs')
((fr pc) # frs) =
(None, hp2, (os2, lvars2, C, S, Suc pc) # frs) ⟧ ⟹
{G, C, S} ⊢ {hp0, os0, lvars0} >-[instr]→ {hp2,os2,lvars2}"
⟨*proof*⟩

lemma *progression_transitive:*
"⟦ instrs_comb = instrs0 @ instrs1;
{G, C, S} ⊢ {hp0, os0, lvars0} >- instrs0 → {hp1, os1, lvars1};
{G, C, S} ⊢ {hp1, os1, lvars1} >- instrs1 → {hp2, os2, lvars2} ⟧
⟹

```
  {G, C, S} ⊢ {hp0, os0, lvars0} >- instrs_comb → {hp2, os2, lvars2}"
```
⟨*proof*⟩

lemma *progression_refl:*
```
  "{G, C, S} ⊢ {hp0, os0, lvars0} >- [] → {hp0, os0, lvars0}"
```
⟨*proof*⟩

lemma *progression_one_step: "*
```
  ∀pc frs.
  (exec_instr i G hp0 os0 lvars0 C S pc frs) =
  (None, hp1, (os1,lvars1,C,S, Suc pc)#frs)
  ⟹ {G, C, S} ⊢ {hp0, os0, lvars0} >- [i] → {hp1, os1, lvars1}"
```
⟨*proof*⟩

definition *jump_fwd :: "jvm_prog ⇒ cname ⇒ sig ⇒*
```
                aheap ⇒ locvars ⇒ opstack ⇒ opstack ⇒
                instr ⇒ bytecode ⇒ bool"  where
  "jump_fwd G C S hp lvars os0 os1 instr instrs ==
  ∀pre post frs.
  (gis (gmb G C S) = pre @ instr # instrs @ post) ⟶
   exec_all G (None,hp,(os0,lvars,C,S, length pre)#frs)
    (None,hp,(os1,lvars,C,S,(length pre) + (length instrs) + 1)#frs)"
```

lemma *jump_fwd_one_step:*
```
  "∀pc frs.
  exec_instr instr G hp os0 lvars C S pc frs =
    (None, hp, (os1, lvars, C, S, pc + (length instrs) + 1)#frs)
  ⟹ jump_fwd G C S hp lvars os0 os1 instr instrs"
```
⟨*proof*⟩

lemma *jump_fwd_progression_aux:*
```
  "⟦ instrs_comb = instr # instrs0 @ instrs1;
     jump_fwd G C S hp lvars os0 os1 instr instrs0;
     {G, C, S} ⊢ {hp, os1, lvars} >- instrs1 → {hp2, os2, lvars2} ⟧
  ⟹ {G, C, S} ⊢ {hp, os0, lvars} >- instrs_comb → {hp2, os2, lvars2}"
```
⟨*proof*⟩

lemma *jump_fwd_progression:*
```
  "⟦ instrs_comb = instr # instrs0 @ instrs1;
  ∀ pc frs.
  exec_instr instr G hp os0 lvars C S pc frs =
    (None, hp, (os1, lvars, C, S, pc + (length instrs0) + 1)#frs);
  {G, C, S} ⊢ {hp, os1, lvars} >- instrs1 → {hp2, os2, lvars2} ⟧
  ⟹ {G, C, S}  ⊢ {hp, os0, lvars} >- instrs_comb → {hp2, os2, lvars2}"
```
⟨*proof*⟩

definition *jump_bwd :: "jvm_prog ⇒ cname ⇒ sig ⇒*
```
                aheap ⇒ locvars ⇒ opstack ⇒ opstack ⇒
```

```
                bytecode ⇒ instr ⇒ bool" where
  "jump_bwd G C S hp lvars os0 os1 instrs instr ==
  ∀ pre post frs.
  (gis (gmb G C S) = pre @ instrs @ instr # post) ⟶
   exec_all G (None,hp,(os0,lvars,C,S, (length pre) + (length instrs))#frs)
    (None,hp,(os1,lvars,C,S, (length pre))#frs)"


lemma jump_bwd_one_step:
  "∀ pc frs.
  exec_instr instr G hp os0 lvars C S (pc + (length instrs)) frs =
    (None, hp, (os1, lvars, C, S, pc)#frs)
  ⟹
  jump_bwd G C S hp lvars os0 os1 instrs instr"
  ⟨proof⟩

lemma jump_bwd_progression:
  "⟦ instrs_comb = instrs @ [instr];
  {G, C, S} ⊢ {hp0, os0, lvars0} >- instrs → {hp1, os1, lvars1};
  jump_bwd G C S hp1 lvars1 os1 os2 instrs instr;
  {G, C, S} ⊢ {hp1, os2, lvars1} >- instrs_comb → {hp3, os3, lvars3} ⟧
  ⟹ {G, C, S}  ⊢ {hp0, os0, lvars0} >- instrs_comb → {hp3, os3, lvars3}"
  ⟨proof⟩




definition class_sig_defined :: "'c prog ⇒ cname ⇒ sig ⇒ bool" where
  "class_sig_defined G C S ==
  is_class G C ∧ (∃ D rT mb. (method (G, C) S = Some (D, rT, mb)))"



definition env_of_jmb :: "java_mb prog ⇒ cname ⇒ sig ⇒ java_mb env" where
  "env_of_jmb G C S ==
  (let (mn,pTs) = S;
       (D,rT,(pns,lvars,blk,res)) = the(method (G, C) S) in
  (G,(map_of lvars)(pns[↦]pTs, This↦Class C)))"

lemma env_of_jmb_fst [simp]: "fst (env_of_jmb G C S) = G"
⟨proof⟩




lemma method_preserves [rule_format (no_asm)]:
  "⟦ wf_prog wf_mb G; is_class G C;
  ∀ S rT mb. ∀ cn ∈ fst ' set G. wf_mdecl wf_mb G cn (S,rT,mb)  ⟶ (P cn S (rT,mb))⟧
 ⟹ ∀ D.
  method (G, C) S = Some (D, rT, mb) ⟶ (P D S (rT,mb))"

  ⟨proof⟩
```

lemma *method_preserves_length:*
"⟦ *wf_java_prog G; is_class G C;*
method (G, C) (mn,pTs) = Some (D, rT, pns, lvars, blk, res)⟧
⟹ *length pns = length pTs*"
⟨*proof*⟩

definition *wtpd_expr :: "java_mb env ⇒ expr ⇒ bool"* **where**
"wtpd_expr E e == (∃ T. E⊢e :: T)"

definition *wtpd_exprs :: "java_mb env ⇒ (expr list) ⇒ bool"* **where**
"wtpd_exprs E e == (∃ T. E⊢e [::] T)"

definition *wtpd_stmt :: "java_mb env ⇒ stmt ⇒ bool"* **where**
"wtpd_stmt E c == (E⊢c √)"

lemma *wtpd_expr_newc: "wtpd_expr E (NewC C) ⟹ is_class (prg E) C"*
⟨*proof*⟩

lemma *wtpd_expr_cast: "wtpd_expr E (Cast cn e) ⟹ (wtpd_expr E e)"*
⟨*proof*⟩

lemma *wtpd_expr_lacc:*
"⟦ *wtpd_expr (env_of_jmb G C S) (LAcc vn); class_sig_defined G C S* ⟧
⟹ *vn ∈ set (gjmb_plns (gmb G C S)) ∨ vn = This*"
⟨*proof*⟩

lemma *wtpd_expr_lass: "wtpd_expr E (vn::=e)*
⟹ *(vn ≠ This) & (wtpd_expr E (LAcc vn)) & (wtpd_expr E e)"*
⟨*proof*⟩

lemma *wtpd_expr_facc: "wtpd_expr E ({fd}a..fn)*
⟹ *(wtpd_expr E a)"*
⟨*proof*⟩

lemma *wtpd_expr_fass: "wtpd_expr E ({fd}a..fn:=v)*
⟹ *(wtpd_expr E ({fd}a..fn)) & (wtpd_expr E v)"*
⟨*proof*⟩

lemma *wtpd_expr_binop: "wtpd_expr E (BinOp bop e1 e2)*
⟹ *(wtpd_expr E e1) & (wtpd_expr E e2)"*
⟨*proof*⟩

lemma *wtpd_exprs_cons: "wtpd_exprs E (e # es)*
⟹ *(wtpd_expr E e) & (wtpd_exprs E es)"*
⟨*proof*⟩

lemma *wtpd_stmt_expr: "wtpd_stmt E (Expr e) ⟹ (wtpd_expr E e)"*
⟨*proof*⟩

lemma *wtpd_stmt_comp: "wtpd_stmt E (s1;; s2) ⟹*
(wtpd_stmt E s1) & (wtpd_stmt E s2)"
⟨*proof*⟩

lemma *wtpd_stmt_cond: "wtpd_stmt E (If(e) s1 Else s2) ⟹*
(wtpd_expr E e) & (wtpd_stmt E s1) & (wtpd_stmt E s2)
& (E⊢e::PrimT Boolean)"
⟨*proof*⟩

lemma *wtpd_stmt_loop: "wtpd_stmt E (While(e) s) ⟹*
(wtpd_expr E e) & (wtpd_stmt E s) & (E⊢e::PrimT Boolean)"
⟨*proof*⟩

lemma *wtpd_expr_call: "wtpd_expr E ({C}a..mn({pTs'}ps))*
⟹ (wtpd_expr E a) & (wtpd_exprs E ps)
& (length ps = length pTs') & (E⊢a::Class C)
& (∃ pTs md rT.
E⊢ps[::]pTs & max_spec (prg E) C (mn, pTs) = {((md,rT),pTs')})"
⟨*proof*⟩

lemma *wtpd_blk:*
"⟦ method (G, D) (md, pTs) = Some (D, rT, (pns, lvars, blk, res));
wf_prog wf_java_mdecl G; is_class G D ⟧
⟹ wtpd_stmt (env_of_jmb G D (md, pTs)) blk"
⟨*proof*⟩

lemma *wtpd_res:*
"⟦ method (G, D) (md, pTs) = Some (D, rT, (pns, lvars, blk, res));
wf_prog wf_java_mdecl G; is_class G D ⟧
⟹ wtpd_expr (env_of_jmb G D (md, pTs)) res"
⟨*proof*⟩

lemma *evals_preserves_length:*
"G⊢ xs -es[≻]vs-> (None, s) ⟹ length es = length vs"
⟨*proof*⟩

lemma *progression_Eq : "{G, C, S} ⊢*
{hp, (v2 # v1 # os), lvars}
>- [Ifcmpeq 3, LitPush (Bool False), Goto 2, LitPush (Bool True)] →
{hp, (Bool (v1 = v2) # os), lvars}"
⟨*proof*⟩

declare *split_paired_All [simp del] split_paired_Ex [simp del]*

lemma *distinct_method:*
"⟦ *wf_java_prog G; is_class G C; method (G, C) S = Some (D, rT, pns, lvars, blk, res)* ⟧ ⟹
distinct (gjmb_plns (gmb G C S))"
⟨*proof*⟩

lemma *distinct_method_if_class_sig_defined :*
"⟦ *wf_java_prog G; class_sig_defined G C S* ⟧ ⟹ *distinct (gjmb_plns (gmb G C S))"*
⟨*proof*⟩

lemma *method_yields_wf_java_mdecl: "⟦ wf_java_prog G; is_class G C;*
method (G, C) S = Some (D, rT, pns, lvars, blk, res) ⟧ ⟹
wf_java_mdecl G D (S,rT,(pns,lvars,blk,res))"
⟨*proof*⟩

lemma *progression_lvar_init_aux [rule_format (no_asm)]: "*
∀ *zs prfx lvals lvars0.*
lvars0 = (zs @ lvars) ⟶
(disjoint_varnames pns lvars0 ⟶
(length lvars = length lvals) ⟶
(Suc(length pns + length zs) = length prfx) ⟶
({cG, D, S} ⊢
{h, os, (prfx @ lvals)}
>- (concat (map (compInit (pns, lvars0, blk, res)) lvars)) →
{h, os, (prfx @ (map (λp. (default_val (snd p))) lvars))}))"
⟨*proof*⟩

lemma *progression_lvar_init [rule_format (no_asm)]:*
"⟦ *wf_java_prog G; is_class G C;*
method (G, C) S = Some (D, rT, pns, lvars, blk, res) ⟧ ⟹
length pns = length pvs ⟶
(∀ *lvals.*
length lvars = length lvals ⟶
{cG, D, S} ⊢
{h, os, (a' # pvs @ lvals)}
>- (compInitLvars (pns, lvars, blk, res) lvars) →
{h, os, (locvars_xstate G C S (Norm (h, (init_vars lvars)(pns[↦]pvs, This↦a'))))})"
⟨*proof*⟩

lemma *state_ok_eval:*
"⟦xs::≼E; wf_java_prog (prg E); wtpd_expr E e; (prg E)⊢xs -e≻v -> xs'⟧ ⟹ xs'::≼E"
⟨*proof*⟩

lemma *state_ok_evals:*
"⟦xs::≼E; wf_java_prog (prg E); wtpd_exprs E es; prg E ⊢ xs -es[≻]vs-> xs'⟧ ⟹ xs'::≼E"
⟨*proof*⟩

lemma *state_ok_exec:*
"⟦xs::≼E; wf_java_prog (prg E); wtpd_stmt E st; prg E ⊢ xs -st-> xs'⟧ ⟹ xs'::≼E"
⟨*proof*⟩

lemma *state_ok_init:*
"⟦ wf_java_prog G; (x, h, l)::≼(env_of_jmb G C S);
is_class G dynT;
method (G, dynT) (mn, pTs) = Some (md, rT, pns, lvars, blk, res);
list_all2 (conf G h) pvs pTs; G,h ⊢ a' ::≼ Class md⟧
⟹
(np a' x, h, (init_vars lvars)(pns[↦]pvs, This↦a'))::≼(env_of_jmb G md (mn, pTs))"
⟨*proof*⟩

lemma *ty_exprs_list_all2 [rule_format (no_asm)]:*
"(∀ Ts. (E ⊢ es [::] Ts) = list_all2 (λe T. E ⊢ e :: T) es Ts)"
⟨*proof*⟩

lemma *conf_bool:* "G,h ⊢ v::≼PrimT Boolean ⟹ ∃ b. v = Bool b"
⟨*proof*⟩

lemma *max_spec_widen:* "max_spec G C (mn, pTs) = {((md,rT),pTs')} ⟹
list_all2 (λ T T'. G ⊢ T ≼ T') pTs pTs'"
⟨*proof*⟩

lemma *eval_conf:* "⟦G ⊢ s -e≻v-> s'; wf_java_prog G; s::≼E;
E⊢e::T; gx s' = None; prg E = G ⟧
⟹ G,gh s'⊢v::≼T"
⟨*proof*⟩

lemma *evals_preserves_conf:*
"⟦ G⊢ s -es[≻]vs-> s'; G,gh s ⊢ t ::≼ T; E ⊢es[::]Ts;
wf_java_prog G; s::≼E;
prg E = G ⟧ ⟹ G,gh s' ⊢ t ::≼ T"
⟨*proof*⟩

lemma *eval_of_class:*
"⟦ G ⊢ s -e≻a'-> s'; E ⊢ e :: Class C; wf_java_prog G; s::≼E; gx s'=None; a' ≠ Null;
G=prg E⟧
⟹ (∃ lc. a' = Addr lc)"
⟨*proof*⟩

lemma *dynT_subcls:*
```
"⟦ a' ≠ Null; G,h⊢a'::≼ Class C; dynT = fst (the (h (the_Addr a')));
is_class G dynT; ws_prog G ⟧ ⟹ G⊢dynT ≼C C"
```
⟨*proof*⟩

lemma *method_defined:*
```
"⟦
  m = the (method (G, dynT) (mn, pTs));
  dynT = fst (the (h a)); is_class G dynT; wf_java_prog G;
  a' ≠ Null; G,h⊢a'::≼ Class C; a = the_Addr a';
  ∃pTsa md rT. max_spec G C (mn, pTsa) = {((md, rT), pTs)} ⟧
⟹ (method (G, dynT) (mn, pTs)) = Some m"
```
⟨*proof*⟩

theorem *compiler_correctness:*
```
"wf_java_prog G ⟹
(G ⊢ xs -ex≻val-> xs' ⟶
gx xs = None ⟶ gx xs' = None ⟶
(∀ os CL S.
(class_sig_defined G CL S) ⟶
(wtpd_expr (env_of_jmb G CL S) ex) ⟶
(xs ::≼ (env_of_jmb G CL S)) ⟶
( {TranslComp.comp G, CL, S} ⊢
  {gh xs, os, (locvars_xstate G CL S xs)}
  >- (compExpr (gmb G CL S) ex) →
  {gh xs', val#os, locvars_xstate G CL S xs'})))) ∧

(G ⊢ xs -exs[≻]vals-> xs' ⟶
gx xs = None ⟶ gx xs' = None ⟶
(∀ os CL S.
(class_sig_defined G CL S) ⟶
(wtpd_exprs (env_of_jmb G CL S) exs) ⟶
(xs::≼(env_of_jmb G CL S)) ⟶
( {TranslComp.comp G, CL, S} ⊢
  {gh xs, os, (locvars_xstate G CL S xs)}
  >- (compExprs (gmb G CL S) exs) →
  {gh xs', (rev vals)@os, (locvars_xstate G CL S xs')}))) ∧

(G ⊢ xs -st-> xs' ⟶
 gx xs = None ⟶ gx xs' = None ⟶
(∀ os CL S.
(class_sig_defined G CL S) ⟶
(wtpd_stmt (env_of_jmb G CL S) st) ⟶
(xs::≼(env_of_jmb G CL S)) ⟶
( {TranslComp.comp G, CL, S} ⊢
  {gh xs, os, (locvars_xstate G CL S xs)}
  >- (compStmt (gmb G CL S) st) →
```

```
    {gh xs', os, (locvars_xstate G CL S xs')}))))"
```
⟨*proof*⟩

theorem *compiler_correctness_eval: "*
```
  ⟦ G ⊢ (None,hp,loc) -ex ≻ val-> (None,hp',loc');
  wf_java_prog G;
  class_sig_defined G C S;
  wtpd_expr (env_of_jmb G C S) ex;
  (None,hp,loc) ::≼ (env_of_jmb G C S) ⟧ ⟹
  {(TranslComp.comp G), C, S} ⊢
    {hp, os, (locvars_locals G C S loc)}
      >- (compExpr (gmb G C S) ex) →
    {hp', val#os, (locvars_locals G C S loc')}"
```
⟨*proof*⟩

theorem *compiler_correctness_exec: "*
```
  ⟦ G ⊢ Norm (hp, loc) -st-> Norm (hp', loc');
  wf_java_prog G;
  class_sig_defined G C S;
  wtpd_stmt (env_of_jmb G C S) st;
  (None,hp,loc) ::≼ (env_of_jmb G C S) ⟧ ⟹
  {(TranslComp.comp G), C, S} ⊢
    {hp, os, (locvars_locals G C S loc)}
      >- (compStmt (gmb G C S) st) →
    {hp', os, (locvars_locals G C S loc')}"
```
⟨*proof*⟩

declare *split_paired_All [simp] split_paired_Ex [simp]*

declare *wf_prog_ws_prog [simp del]*

end

theory *TypeInf*
imports *"../J/WellType"*
begin

lemma *NewC_invers:*
assumes *"E⊢NewC C::T"*
shows *"T = Class C ∧ is_class (prg E) C"*

⟨*proof*⟩

lemma *Cast_invers:*
assumes *"E⊢Cast D e::T"*
shows *"∃C. T = Class D ∧ E⊢e::C ∧ is_class (prg E) D ∧ prg E⊢C⪯? Class D"*
⟨*proof*⟩

lemma *Lit_invers:*
assumes *"E⊢Lit x::T"*
shows *"typeof (λv. None) x = Some T"*
⟨*proof*⟩

lemma *LAcc_invers:*
assumes *"E⊢LAcc v::T"*
shows *"localT E v = Some T ∧ is_type (prg E) T"*
⟨*proof*⟩

lemma *BinOp_invers:*
assumes *"E⊢BinOp bop e1 e2::T'"*
shows *"∃T. E⊢e1::T ∧ E⊢e2::T ∧*
(if bop = Eq then T' = PrimT Boolean
else T' = T ∧ T = PrimT Integer)"
⟨*proof*⟩

lemma *LAss_invers:*
assumes *"E⊢v::=e::T'"*
shows *"∃T. v ~= This ∧ E⊢LAcc v::T ∧ E⊢e::T' ∧ prg E⊢T'⪯T"*
⟨*proof*⟩

lemma *FAcc_invers:*
assumes *"E⊢{fd}a..fn::fT"*
shows *"∃C. E⊢a::Class C ∧ field (prg E,C) fn = Some (fd, fT)"*
⟨*proof*⟩

lemma *FAss_invers:*
assumes *"E⊢{fd}a..fn:=v::T'"*
shows *"∃T. E⊢{fd}a..fn::T ∧ E⊢v ::T' ∧ prg E⊢T'⪯T"*
⟨*proof*⟩

lemma *Call_invers:*
assumes *"E⊢{C}a..mn({pTs'}ps)::rT"*
shows *"∃pTs md.*
E⊢a::Class C ∧ E⊢ps[::]pTs ∧ max_spec (prg E) C (mn, pTs) = {((md,rT),pTs')}"
⟨*proof*⟩

lemma *Nil_invers:*
assumes *"E⊢[] [::] Ts"*
shows *"Ts = []"*
⟨*proof*⟩

lemma *Cons_invers:*
assumes *"E⊢e#es[::]Ts"*
shows *"∃T Ts'. Ts = T#Ts' ∧ E ⊢e::T ∧ E ⊢es[::]Ts'"*

⟨*proof*⟩

lemma *Expr_invers:*
assumes *"E⊢Expr e√"*
shows *"∃ T. E⊢e::T"*
⟨*proof*⟩

lemma *Comp_invers:*
assumes *"E⊢s1;; s2√"*
shows *"E⊢s1√ ∧ E⊢s2√"*
⟨*proof*⟩

lemma *Cond_invers:*
assumes *"E⊢If(e) s1 Else s2√"*
shows *"E⊢e::PrimT Boolean ∧ E⊢s1√ ∧ E⊢s2√"*
⟨*proof*⟩

lemma *Loop_invers:*
assumes *"E⊢While(e) s√"*
shows *"E⊢e::PrimT Boolean ∧ E⊢s√"*
⟨*proof*⟩

declare *split_paired_All [simp del]*
declare *split_paired_Ex [simp del]*

method *ty_case_simp = ((erule ty_exprs.cases ty_expr.cases; simp)+)[]*
method *strip_case_simp = (intro strip, ty_case_simp)*

lemma *uniqueness_of_types: "*
(∀ (E::'a prog × (vname ⇒ ty option)) T1 T2.
E⊢e :: T1 ⟶ E⊢e :: T2 ⟶ T1 = T2) ∧
(∀ (E::'a prog × (vname ⇒ ty option)) Ts1 Ts2.
E⊢es [::] Ts1 ⟶ E⊢es [::] Ts2 ⟶ Ts1 = Ts2)"
⟨*proof*⟩

lemma *uniqueness_of_types_expr [rule_format (no_asm)]: "*
(∀E T1 T2. E⊢e :: T1 ⟶ E⊢e :: T2 ⟶ T1 = T2)"
⟨*proof*⟩

lemma *uniqueness_of_types_exprs [rule_format (no_asm)]: "*
(∀E Ts1 Ts2. E⊢es [::] Ts1 ⟶ E⊢es [::] Ts2 ⟶ Ts1 = Ts2)"
⟨*proof*⟩

definition *inferred_tp :: "[java_mb env, expr] ⇒ ty"* **where**
"inferred_tp E e == (SOME T. E⊢e :: T)"

definition *inferred_tps :: "[java_mb env, expr list] ⇒ ty list"* **where**
"inferred_tps E es == (SOME Ts. E⊢es [::] Ts)"

lemma *inferred_tp_wt: "E⊢e :: T ⟹ (inferred_tp E e) = T"*
⟨*proof*⟩

lemma *inferred_tps_wt: "E⊢es [::] Ts ⟹ (inferred_tps E es) = Ts"*
⟨*proof*⟩

end

4.27 Alternative definition of well-typing of bytecode, used in compiler type correctness proof

theory *Altern*
imports *BVSpec*
begin

definition *check_type :: "jvm_prog ⇒ nat ⇒ nat ⇒ JVMType.state ⇒ bool"* **where**
"check_type G mxs mxr s ≡ s ∈ states G mxs mxr"

definition *wt_instr_altern :: "[instr,jvm_prog,ty,method_type,nat,nat,p_count,*
exception_table,p_count] ⇒ bool" **where**
"wt_instr_altern i G rT phi mxs mxr max_pc et pc ≡
app i G mxs rT pc et (phi!pc) ∧
check_type G mxs mxr (OK (phi!pc)) ∧
(∀(pc',s') ∈ set (eff i G pc et (phi!pc)). pc' < max_pc ∧ G ⊢ s' <=' phi!pc')"

definition *wt_method_altern :: "[jvm_prog,cname,ty list,ty,nat,nat,instr list,*
exception_table,method_type] ⇒ bool" **where**
"wt_method_altern G C pTs rT mxs mxl ins et phi ≡
let max_pc = length ins in
0 < max_pc ∧
length phi = length ins ∧
check_bounded ins et ∧
wt_start G C pTs mxl phi ∧
(∀pc. pc<max_pc ⟶ wt_instr_altern (ins!pc) G rT phi mxs (1+length pTs+mxl) max_pc
et pc)"

lemma *wt_method_wt_method_altern :*
"wt_method G C pTs rT mxs mxl ins et phi ⟶ wt_method_altern G C pTs rT mxs mxl ins
et phi"
⟨*proof*⟩

lemma *check_type_check_types [rule_format]:*
"(∀pc. pc < length phi ⟶ check_type G mxs mxr (OK (phi ! pc)))
⟶ check_types G mxs mxr (map OK phi)"
⟨*proof*⟩

lemma *wt_method_altern_wt_method [rule_format]:*
"wt_method_altern G C pTs rT mxs mxl ins et phi ⟶ wt_method G C pTs rT mxs mxl ins et phi"
⟨*proof*⟩

end

theory *CorrCompTp*
imports *LemmasComp TypeInf "../BV/JVM" "../BV/Altern"*
begin

declare *split_paired_All [simp del]*
declare *split_paired_Ex [simp del]*

definition *inited_LT :: "[cname, ty list, (vname × ty) list] ⇒ locvars_type"* **where**
"inited_LT C pTs lvars == (OK (Class C))#((map OK pTs))@(map (Fun.comp OK snd) lvars)"

definition *is_inited_LT :: "[cname, ty list, (vname × ty) list, locvars_type] ⇒ bool"* **where**
"is_inited_LT C pTs lvars LT == (LT = (inited_LT C pTs lvars))"

definition *local_env :: "[java_mb prog, cname, sig, vname list,(vname × ty) list] ⇒ java_mb env"* **where**
"local_env G C S pns lvars ==
let (mn, pTs) = S in (G,(map_of lvars)(pns[↦]pTs, This↦Class C))"

lemma *local_env_fst [simp]: "fst (local_env G C S pns lvars) = G"*
⟨*proof*⟩

lemma *wt_class_expr_is_class:*
"⟦ ws_prog G; E ⊢ expr :: Class cname; E = local_env G C (mn, pTs) pns lvars⟧
⟹ is_class G cname "
⟨*proof*⟩

4.27.1 index

lemma *local_env_snd:*
"snd (local_env G C (mn, pTs) pns lvars) = (map_of lvars)(pns[↦]pTs, This↦Class C)"
⟨*proof*⟩

lemma *index_in_bounds:*
"length pns = length pTs ⟹
snd (local_env G C (mn, pTs) pns lvars) vname = Some T
⟹ index (pns, lvars, blk, res) vname < length (inited_LT C pTs lvars)"
⟨*proof*⟩

lemma *map_upds_append:*
"length k1s = length x1s ⟹ m(k1s[↦]x1s, k2s[↦]x2s) = m ((k1s@k2s)[↦](x1s@x2s))"
⟨*proof*⟩

lemma *map_of_append:*
"map_of ((rev xs) @ ys) = (map_of ys) ((map fst xs) [↦] (map snd xs))"
⟨*proof*⟩

lemma *map_of_as_map_upds: "map_of (rev xs) = Map.empty ((map fst xs) [↦] (map snd xs))"*
⟨*proof*⟩

lemma *map_of_rev: "unique xs ⟹ map_of (rev xs) = map_of xs"*
⟨*proof*⟩

lemma *map_upds_rev:*
"⟦ distinct ks; length ks = length xs ⟧ ⟹ m (rev ks [↦] rev xs) = m (ks [↦] xs)"
⟨*proof*⟩

lemma *map_upds_takeWhile [rule_format]:*
"∀ks. (Map.empty(rev ks[↦]rev xs)) k = Some x ⟶ length ks = length xs ⟶
xs ! length (takeWhile (λz. z ≠ k) ks) = x"
⟨*proof*⟩

lemma *local_env_inited_LT:*
"⟦ snd (local_env G C (mn, pTs) pns lvars) vname = Some T;
length pns = length pTs; distinct pns; unique lvars ⟧
⟹ (inited_LT C pTs lvars ! index (pns, lvars, blk, res) vname) = OK T"
⟨*proof*⟩

lemma *inited_LT_at_index_no_err:*
"i < length (inited_LT C pTs lvars) ⟹ inited_LT C pTs lvars ! i ≠ Err"
⟨*proof*⟩

lemma *sup_loc_update_index: "*
⟦ G ⊢ T ≼ T'; is_type G T'; length pns = length pTs; distinct pns; unique lvars;
snd (local_env G C (mn, pTs) pns lvars) vname = Some T' ⟧
⟹
comp G ⊢ (inited_LT C pTs lvars) [index (pns, lvars, blk, res) vname := OK T] <=l
inited_LT C pTs lvars"
⟨*proof*⟩

4.27.2 Preservation of ST and LT by compTpExpr / compTpStmt

lemma *sttp_of_comb_nil [simp]: "sttp_of (comb_nil sttp) = sttp"*
⟨*proof*⟩

lemma *mt_of_comb_nil [simp]: "mt_of (comb_nil sttp) = []"*
⟨*proof*⟩

lemma *sttp_of_comb [simp]: "sttp_of ((f1 □ f2) sttp) = sttp_of (f2 (sttp_of (f1 sttp)))"*
⟨*proof*⟩

lemma *mt_of_comb: "(mt_of ((f1 □ f2) sttp)) =*
(mt_of (f1 sttp)) @ (mt_of (f2 (sttp_of (f1 sttp))))"
⟨*proof*⟩

lemma *mt_of_comb_length [simp]: "⟦ n1 = length (mt_of (f1 sttp)); n1 ≤ n ⟧*
⟹ (mt_of ((f1 □ f2) sttp) ! n) = (mt_of (f2 (sttp_of (f1 sttp))) ! (n - n1))"
⟨*proof*⟩

lemma *compTpExpr_Exprs_LT_ST: "*
⟦jmb = (pns, lvars, blk, res);
wf_prog wf_java_mdecl G;
wf_java_mdecl G C ((mn, pTs), rT, jmb);
E = local_env G C (mn, pTs) pns lvars ⟧
⟹
(∀ ST LT T.
E ⊢ ex :: T ⟶
is_inited_LT C pTs lvars LT ⟶
sttp_of (compTpExpr jmb G ex (ST, LT)) = (T # ST, LT))
∧
(∀ ST LT Ts.
E ⊢ exs [::] Ts ⟶
is_inited_LT C pTs lvars LT ⟶
sttp_of (compTpExprs jmb G exs (ST, LT)) = ((rev Ts) @ ST, LT))"

⟨*proof*⟩

lemmas *compTpExpr_LT_ST [rule_format (no_asm)] =*
compTpExpr_Exprs_LT_ST [THEN conjunct1]

lemmas *compTpExprs_LT_ST [rule_format (no_asm)] =*
compTpExpr_Exprs_LT_ST [THEN conjunct2]

lemma *compTpStmt_LT_ST [rule_format (no_asm)]: "*
⟦ jmb = (pns,lvars,blk,res);
wf_prog wf_java_mdecl G;
wf_java_mdecl G C ((mn, pTs), rT, jmb);
E = (local_env G C (mn, pTs) pns lvars)⟧
⟹ (∀ ST LT.
E ⊢ s√ ⟶
(is_inited_LT C pTs lvars LT)
⟶ sttp_of (compTpStmt jmb G s (ST, LT)) = (ST, LT))"

⟨*proof*⟩

lemma *compTpInit_LT_ST: "*
sttp_of (compTpInit jmb (vn,ty) (ST, LT)) = (ST, LT[(index jmb vn):= OK ty])"
⟨*proof*⟩

lemma *compTpInitLvars_LT_ST_aux [rule_format (no_asm)]:*
"∀ pre lvars_pre lvars0.
jmb = (pns,lvars0,blk,res) ∧
lvars0 = (lvars_pre @ lvars) ∧
(length pns) + (length lvars_pre) + 1 = length pre ∧
disjoint_varnames pns (lvars_pre @ lvars)
⟶
sttp_of (compTpInitLvars jmb lvars (ST, pre @ replicate (length lvars) Err))
= (ST, pre @ map (Fun.comp OK snd) lvars)"
⟨*proof*⟩

lemma *compTpInitLvars_LT_ST:*
"⟦ jmb = (pns, lvars, blk, res); wf_java_mdecl G C ((mn, pTs), rT, jmb) ⟧
⟹ sttp_of (compTpInitLvars jmb lvars (ST, start_LT C pTs (length lvars)))
= (ST, inited_LT C pTs lvars)"
⟨*proof*⟩

lemma *max_of_list_elem: "x ∈ set xs ⟹ x ≤ (max_of_list xs)"*
⟨*proof*⟩

lemma *max_of_list_sublist: "set xs ⊆ set ys*
⟹ (max_of_list xs) ≤ (max_of_list ys)"
⟨*proof*⟩

lemma *max_of_list_append [simp]:*
"max_of_list (xs @ ys) = max (max_of_list xs) (max_of_list ys)"
⟨*proof*⟩

lemma *app_mono_mxs: "⟦ app i G mxs rT pc et s; mxs ≤ mxs' ⟧*
⟹ app i G mxs' rT pc et s"
⟨*proof*⟩

lemma *err_mono [simp]: "A ⊆ B ⟹ err A ⊆ err B"*
⟨*proof*⟩

lemma *opt_mono [simp]: "A ⊆ B ⟹ opt A ⊆ opt B"*
⟨*proof*⟩

lemma *states_mono: "⟦ mxs ≤ mxs' ⟧*
⟹ states G mxs mxr ⊆ states G mxs' mxr"
⟨*proof*⟩

lemma *check_type_mono:*
"⟦ check_type G mxs mxr s; mxs ≤ mxs' ⟧ ⟹ check_type G mxs' mxr s"
⟨*proof*⟩

lemma *wt_instr_prefix: "*
⟦ wt_instr_altern (bc ! pc) cG rT mt mxs mxr max_pc et pc;
bc' = bc @ bc_post; mt' = mt @ mt_post;
mxs ≤ mxs'; max_pc ≤ max_pc';
pc < length bc; pc < length mt;
max_pc = (length mt)⟧
⟹ wt_instr_altern (bc' ! pc) cG rT mt' mxs' mxr max_pc' et pc"
⟨*proof*⟩

lemma *pc_succs_shift:*
"pc'∈set (succs i (pc'' + n)) ⟹ ((pc' - n) ∈set (succs i pc''))"
⟨*proof*⟩

lemma *pc_succs_le:*
"⟦ pc' ∈ set (succs i (pc'' + n));
∀ b. ((i = (Goto b) ∨ i=(Ifcmpeq b)) ⟶ 0 ≤ (int pc'' + b)) ⟧
⟹ n ≤ pc'"
⟨*proof*⟩

definition *offset_xcentry :: "[nat, exception_entry] ⇒ exception_entry"* **where**
"offset_xcentry ==
λ n (start_pc, end_pc, handler_pc, catch_type).
(start_pc + n, end_pc + n, handler_pc + n, catch_type)"

definition *offset_xctable :: "[nat, exception_table] ⇒ exception_table"* **where**
"offset_xctable n == (map (offset_xcentry n))"

lemma *match_xcentry_offset [simp]: "*
match_exception_entry G cn (pc + n) (offset_xcentry n ee) =
match_exception_entry G cn pc ee"
⟨*proof*⟩

lemma *match_xctable_offset: "*
(match_exception_table G cn (pc + n) (offset_xctable n et)) =
(map_option (λ pc'. pc' + n) (match_exception_table G cn pc et))"
⟨*proof*⟩

lemma *match_offset [simp]: "*
match G cn (pc + n) (offset_xctable n et) = match G cn pc et"
⟨*proof*⟩

lemma *match_any_offset [simp]: "*
match_any G (pc + n) (offset_xctable n et) = match_any G pc et"
⟨*proof*⟩

lemma *app_mono_pc: "⟦ app i G mxs rT pc et s; pc'= pc + n ⟧*
⟹ app i G mxs rT pc' (offset_xctable n et) s"
⟨*proof*⟩

abbreviation *(input)*
empty_et :: exception_table
where *"empty_et == []"*

lemma *xcpt_names_Nil [simp]: "(xcpt_names (i, G, pc, [])) = []"*
⟨*proof*⟩

lemma *xcpt_eff_Nil [simp]: "(xcpt_eff i G pc s []) = []"*
⟨*proof*⟩

lemma *app_jumps_lem: "⟦ app i cG mxs rT pc empty_et s; s=(Some st) ⟧*
⟹ ∀ b. ((i = (Goto b) ∨ i=(Ifcmpeq b)) ⟶ 0 ≤ (int pc + b))"
⟨*proof*⟩

lemma *wt_instr_offset: "*
⟦ ∀ pc'' < length mt.
wt_instr_altern ((bc@bc_post) ! pc'') cG rT (mt@mt_post) mxs mxr max_pc empty_et pc'';

bc' = bc_pre @ bc @ bc_post; mt' = mt_pre @ mt @ mt_post;
length bc_pre = length mt_pre; length bc = length mt;
length mt_pre ≤ pc; pc < length (mt_pre @ mt);
mxs ≤ mxs'; max_pc + length mt_pre ≤ max_pc' ⟧
⟹ wt_instr_altern (bc' ! pc) cG rT mt' mxs' mxr max_pc' empty_et pc"
⟨*proof*⟩

definition *start_sttp_resp_cons :: "[state_type ⇒ method_type × state_type] ⇒ bool"* **where**
"start_sttp_resp_cons f ==

```
    (∀ sttp. let (mt', sttp') = (f sttp) in (∃mt'_rest. mt' = Some sttp # mt'_rest))"
```

definition *start_sttp_resp :: "[state_type ⇒ method_type × state_type] ⇒ bool"* **where**
"start_sttp_resp f == (f = comb_nil) ∨ (start_sttp_resp_cons f)"

lemma *start_sttp_resp_comb_nil [simp]: "start_sttp_resp comb_nil"*
⟨*proof*⟩

lemma *start_sttp_resp_cons_comb_cons [simp]: "start_sttp_resp_cons f*
⟹ start_sttp_resp_cons (f □ f')"
⟨*proof*⟩

lemma *start_sttp_resp_cons_comb_cons_r: "⟦ start_sttp_resp f; start_sttp_resp_cons f'⟧*
⟹ start_sttp_resp_cons (f □ f')"
⟨*proof*⟩

lemma *start_sttp_resp_cons_comb [simp]: "start_sttp_resp_cons f*
⟹ start_sttp_resp (f □ f')"
⟨*proof*⟩

lemma *start_sttp_resp_comb: "⟦ start_sttp_resp f; start_sttp_resp f' ⟧*
⟹ start_sttp_resp (f □ f')"
⟨*proof*⟩

lemma *start_sttp_resp_cons_nochangeST [simp]: "start_sttp_resp_cons nochangeST"*
⟨*proof*⟩

lemma *start_sttp_resp_cons_pushST [simp]: "start_sttp_resp_cons (pushST Ts)"*
⟨*proof*⟩

lemma *start_sttp_resp_cons_dupST [simp]: "start_sttp_resp_cons dupST"*
⟨*proof*⟩

lemma *start_sttp_resp_cons_dup_x1ST [simp]: "start_sttp_resp_cons dup_x1ST"*
⟨*proof*⟩

lemma *start_sttp_resp_cons_popST [simp]: "start_sttp_resp_cons (popST n)"*
⟨*proof*⟩

lemma *start_sttp_resp_cons_replST [simp]: "start_sttp_resp_cons (replST n tp)"*
⟨*proof*⟩

lemma *start_sttp_resp_cons_storeST [simp]: "start_sttp_resp_cons (storeST i tp)"*
⟨*proof*⟩

lemma *start_sttp_resp_cons_compTpExpr [simp]: "start_sttp_resp_cons (compTpExpr jmb G ex)"*
⟨*proof*⟩

lemma *start_sttp_resp_cons_compTpInit [simp]: "start_sttp_resp_cons (compTpInit jmb lv)"*
⟨*proof*⟩

lemma *start_sttp_resp_nochangeST [simp]: "start_sttp_resp nochangeST"*

⟨*proof*⟩

lemma *start_sttp_resp_pushST [simp]: "start_sttp_resp (pushST Ts)"*
⟨*proof*⟩

lemma *start_sttp_resp_dupST [simp]: "start_sttp_resp dupST"*
⟨*proof*⟩

lemma *start_sttp_resp_dup_x1ST [simp]: "start_sttp_resp dup_x1ST"*
⟨*proof*⟩

lemma *start_sttp_resp_popST [simp]: "start_sttp_resp (popST n)"*
⟨*proof*⟩

lemma *start_sttp_resp_replST [simp]: "start_sttp_resp (replST n tp)"*
⟨*proof*⟩

lemma *start_sttp_resp_storeST [simp]: "start_sttp_resp (storeST i tp)"*
⟨*proof*⟩

lemma *start_sttp_resp_compTpExpr [simp]: "start_sttp_resp (compTpExpr jmb G ex)"*
⟨*proof*⟩

lemma *start_sttp_resp_compTpExprs [simp]: "start_sttp_resp (compTpExprs jmb G exs)"*
⟨*proof*⟩

lemma *start_sttp_resp_compTpStmt [simp]: "start_sttp_resp (compTpStmt jmb G s)"*
⟨*proof*⟩

lemma *start_sttp_resp_compTpInitLvars [simp]: "start_sttp_resp (compTpInitLvars jmb lvars)"*
⟨*proof*⟩

4.27.3 length of compExpr/ compTpExprs

lemma *length_comb [simp]: "length (mt_of ((f1 □ f2) sttp)) =*
length (mt_of (f1 sttp)) + length (mt_of (f2 (sttp_of (f1 sttp))))"
⟨*proof*⟩

lemma *length_comb_nil [simp]: "length (mt_of (comb_nil sttp)) = 0"*
⟨*proof*⟩

lemma *length_nochangeST [simp]: "length (mt_of (nochangeST sttp)) = 1"*
⟨*proof*⟩

lemma *length_pushST [simp]: "length (mt_of (pushST Ts sttp)) = 1"*
⟨*proof*⟩

lemma *length_dupST [simp]: "length (mt_of (dupST sttp)) = 1"*
⟨*proof*⟩

lemma *length_dup_x1ST [simp]: "length (mt_of (dup_x1ST sttp)) = 1"*
⟨*proof*⟩

lemma *length_popST [simp]: "length (mt_of (popST n sttp)) = 1"*
⟨*proof*⟩

lemma *length_replST [simp]: "length (mt_of (replST n tp sttp)) = 1"*
⟨*proof*⟩

lemma *length_storeST [simp]: "length (mt_of (storeST i tp sttp)) = 1"*
⟨*proof*⟩

lemma *length_compTpExpr_Exprs [rule_format]: "*
(∀sttp. (length (mt_of (compTpExpr jmb G ex sttp)) = length (compExpr jmb ex)))
∧ (∀sttp. (length (mt_of (compTpExprs jmb G exs sttp)) = length (compExprs jmb exs)))"
⟨*proof*⟩

lemma *length_compTpExpr: "length (mt_of (compTpExpr jmb G ex sttp)) = length (compExpr jmb ex)"*
⟨*proof*⟩

lemma *length_compTpExprs: "length (mt_of (compTpExprs jmb G exs sttp)) = length (compExprs jmb exs)"*
⟨*proof*⟩

lemma *length_compTpStmt [rule_format]: "*
(∀ sttp. (length (mt_of (compTpStmt jmb G s sttp)) = length (compStmt jmb s)))"
⟨*proof*⟩

lemma *length_compTpInit: "length (mt_of (compTpInit jmb lv sttp)) = length (compInit jmb lv)"*
⟨*proof*⟩

lemma *length_compTpInitLvars [rule_format]:*
"∀ sttp. length (mt_of (compTpInitLvars jmb lvars sttp)) = length (compInitLvars jmb lvars)"
⟨*proof*⟩

4.27.4 Correspondence bytecode - method types

abbreviation *(input)*
ST_of :: "state_type ⇒ opstack_type"
where *"ST_of == fst"*

abbreviation *(input)*
LT_of :: "state_type ⇒ locvars_type"
where *"LT_of == snd"*

lemma *states_lower:*
"⟦ OK (Some (ST, LT)) ∈ states cG mxs mxr; length ST ≤ mxs⟧
⟹ OK (Some (ST, LT)) ∈ states cG (length ST) mxr"
⟨*proof*⟩

lemma *check_type_lower:*
"⟦ check_type cG mxs mxr (OK (Some (ST, LT))); length ST ≤ mxs⟧
⟹check_type cG (length ST) mxr (OK (Some (ST, LT)))"

⟨*proof*⟩

definition *bc_mt_corresp :: "*
[bytecode, state_type ⇒ method_type × state_type, state_type, jvm_prog, ty, nat, p_count]
⇒ bool" **where**

"bc_mt_corresp bc f sttp0 cG rT mxr idx ==
let (mt, sttp) = f sttp0 in
(length bc = length mt ∧
((check_type cG (length (ST_of sttp0)) mxr (OK (Some sttp0))) ⟶
(∀ mxs.
mxs = max_ssize (mt@[Some sttp]) ⟶
(∀ pc. pc < idx ⟶
wt_instr_altern (bc ! pc) cG rT (mt@[Some sttp]) mxs mxr (length mt + 1) empty_et
pc)
∧
check_type cG mxs mxr (OK ((mt@[Some sttp]) ! idx)))))"

lemma *bc_mt_corresp_comb: "*
⟦ bc' = (bc1@bc2); l' = (length bc');
bc_mt_corresp bc1 f1 sttp0 cG rT mxr (length bc1);
bc_mt_corresp bc2 f2 (sttp_of (f1 sttp0)) cG rT mxr (length bc2);
start_sttp_resp f2⟧
⟹ bc_mt_corresp bc' (f1 □ f2) sttp0 cG rT mxr l'"
⟨*proof*⟩

lemma *bc_mt_corresp_zero [simp]:*
"⟦ length (mt_of (f sttp)) = length bc; start_sttp_resp f⟧
⟹ bc_mt_corresp bc f sttp cG rT mxr 0"
⟨*proof*⟩

definition *mt_sttp_flatten :: "method_type × state_type ⇒ method_type"* **where**
"mt_sttp_flatten mt_sttp == (mt_of mt_sttp) @ [Some (sttp_of mt_sttp)]"

lemma *mt_sttp_flatten_length [simp]: "n = (length (mt_of (f sttp)))*
⟹ (mt_sttp_flatten (f sttp)) ! n = Some (sttp_of (f sttp))"
⟨*proof*⟩

lemma *mt_sttp_flatten_comb: "(mt_sttp_flatten ((f1 □ f2) sttp)) =*
(mt_of (f1 sttp)) @ (mt_sttp_flatten (f2 (sttp_of (f1 sttp))))"
⟨*proof*⟩

lemma *mt_sttp_flatten_comb_length [simp]: "⟦ n1 = length (mt_of (f1 sttp)); n1 ≤ n ⟧*
⟹ (mt_sttp_flatten ((f1 □ f2) sttp) ! n) = (mt_sttp_flatten (f2 (sttp_of (f1 sttp)))
! (n - n1))"
⟨*proof*⟩

lemma *mt_sttp_flatten_comb_zero [simp]:*
"start_sttp_resp f ⟹ (mt_sttp_flatten (f sttp)) ! 0 = Some sttp"
⟨*proof*⟩

lemma *int_outside_right: "0 ≤ (m::int) ⟹ m + (int n) = int ((nat m) + n)"*
⟨*proof*⟩

lemma *int_outside_left: "0 ≤ (m::int) ⟹ (int n) + m = int (n + (nat m))"*
⟨*proof*⟩

lemma *less_Suc [simp] : "n ≤ k ⟹ (k < Suc n) = (k = n)"*
⟨*proof*⟩

lemmas *check_type_simps = check_type_def states_def JVMType.sl_def*
Product.esl_def stk_esl_def reg_sl_def upto_esl_def Listn.sl_def Err.sl_def
JType.esl_def Err.esl_def Err.le_def Listn.le_def Product.le_def Product.sup_def Err.sup_def
Opt.esl_def Listn.sup_def

lemma *check_type_push:*
"⟦ is_class cG cname; check_type cG (length ST) mxr (OK (Some (ST, LT))) ⟧
⟹ check_type cG (Suc (length ST)) mxr (OK (Some (Class cname # ST, LT)))"
⟨*proof*⟩

lemma *bc_mt_corresp_New: "⟦is_class cG cname ⟧*
⟹ bc_mt_corresp [New cname] (pushST [Class cname]) (ST, LT) cG rT mxr (Suc 0)"
⟨*proof*⟩

lemma *bc_mt_corresp_Pop: "*
bc_mt_corresp [Pop] (popST (Suc 0)) (T # ST, LT) cG rT mxr (Suc 0)"
⟨*proof*⟩

lemma *bc_mt_corresp_Checkcast: "⟦ is_class cG cname; sttp = (ST, LT);*
(∃rT STo. ST = RefT rT # STo) ⟧
⟹ bc_mt_corresp [Checkcast cname] (replST (Suc 0) (Class cname)) sttp cG rT mxr (Suc
0)"
⟨*proof*⟩

lemma *bc_mt_corresp_LitPush: "⟦ typeof (λv. None) val = Some T ⟧*
⟹ bc_mt_corresp [LitPush val] (pushST [T]) sttp cG rT mxr (Suc 0)"
⟨*proof*⟩

lemma *bc_mt_corresp_LitPush_CT:*
"⟦ typeof (λv. None) val = Some T ∧ cG ⊢ T ≼ T'; is_type cG T' ⟧
⟹ bc_mt_corresp [LitPush val] (pushST [T']) sttp cG rT mxr (Suc 0)"
⟨*proof*⟩

declare *not_Err_eq [iff del]*

lemma *bc_mt_corresp_Load: "⟦ i < length LT; LT ! i ≠ Err; mxr = length LT ⟧*
⟹ bc_mt_corresp [Load i]
(λ(ST, LT). pushST [ok_val (LT ! i)] (ST, LT)) (ST, LT) cG rT mxr (Suc 0)"
⟨*proof*⟩

lemma *bc_mt_corresp_Store_init:*
"i < length LT ⟹ bc_mt_corresp [Store i] (storeST i T) (T # ST, LT) cG rT mxr (Suc 0)"
⟨*proof*⟩

lemma *bc_mt_corresp_Store:*
"⟦ i < length LT; cG ⊢ LT[i := OK T] <=l LT ⟧
⟹ bc_mt_corresp [Store i] (popST (Suc 0)) (T # ST, LT) cG rT mxr (Suc 0)"
⟨*proof*⟩

lemma *bc_mt_corresp_Dup: "*
bc_mt_corresp [Dup] dupST (T # ST, LT) cG rT mxr (Suc 0)"
⟨*proof*⟩

lemma *bc_mt_corresp_Dup_x1: "*
bc_mt_corresp [Dup_x1] dup_x1ST (T1 # T2 # ST, LT) cG rT mxr (Suc 0)"
⟨*proof*⟩

lemma *bc_mt_corresp_IAdd: "*
bc_mt_corresp [IAdd] (replST 2 (PrimT Integer))
(PrimT Integer # PrimT Integer # ST, LT) cG rT mxr (Suc 0)"
⟨*proof*⟩

lemma *bc_mt_corresp_Getfield: "⟦ wf_prog wf_mb G;*
field (G, C) vname = Some (cname, T); is_class G C ⟧
⟹ bc_mt_corresp [Getfield vname cname]
(replST (Suc 0) (snd (the (field (G, cname) vname))))
(Class C # ST, LT) (comp G) rT mxr (Suc 0)"
⟨*proof*⟩

lemma *bc_mt_corresp_Putfield: "⟦ wf_prog wf_mb G;*
field (G, C) vname = Some (cname, Ta); G ⊢ T ≼ Ta; is_class G C ⟧
⟹ bc_mt_corresp [Putfield vname cname] (popST 2) (T # Class C # T # ST, LT)
(comp G) rT mxr (Suc 0)"
⟨*proof*⟩

lemma *Call_app:*
```
  "⟦ wf_prog wf_mb G; is_class G cname;
     STs = rev pTsa @ Class cname # ST;
     max_spec G cname (mname, pTsa) = {((md, T), pTs')} ⟧
  ⟹ app (Invoke cname mname pTs') (comp G) (length (T # ST)) rT 0 empty_et (Some (STs,
LTs))"
```
⟨*proof*⟩

lemma *bc_mt_corresp_Invoke:*
```
  "⟦ wf_prog wf_mb G;
  max_spec G cname (mname, pTsa) = {((md, T), fpTs)};
  is_class G cname ⟧
 ⟹ bc_mt_corresp [Invoke cname mname fpTs] (replST (Suc (length pTsa)) T)
          (rev pTsa @ Class cname # ST, LT) (comp G) rT mxr (Suc 0)"
```
⟨*proof*⟩

lemma *wt_instr_Ifcmpeq:* "⟦*Suc pc < max_pc;*
```
  0 ≤ (int pc + i);  nat (int pc + i) < max_pc;
  (mt_sttp_flatten f ! pc = Some (ts#ts'#ST,LT)) ∧
  ((∃p. ts = PrimT p ∧ ts' = PrimT p) ∨ (∃r r'. ts = RefT r ∧ ts' = RefT r'));
  mt_sttp_flatten f ! Suc pc = Some (ST,LT);
  mt_sttp_flatten f ! nat (int pc + i) = Some (ST,LT);
  check_type (TranslComp.comp G) mxs mxr (OK (Some (ts # ts' # ST, LT))) ⟧
    ⟹  wt_instr_altern (Ifcmpeq i) (comp G) rT  (mt_sttp_flatten f) mxs mxr max_pc empty_et
pc"
```
⟨*proof*⟩

lemma *wt_instr_Goto:* "⟦ *0 ≤ (int pc + i); nat (int pc + i) < max_pc;*
```
  mt_sttp_flatten f ! nat (int pc + i) = (mt_sttp_flatten f ! pc);
  check_type (TranslComp.comp G) mxs mxr (OK (mt_sttp_flatten f ! pc)) ⟧
  ⟹ wt_instr_altern (Goto i) (comp G) rT  (mt_sttp_flatten f) mxs mxr max_pc empty_et
pc"
```
⟨*proof*⟩

lemma *bc_mt_corresp_comb_inside: "*
```
  ⟦
  bc_mt_corresp bc' f' sttp0 cG rT mxr l1;
  bc' = (bc1@bc2@bc3); f'= (f1 □ f2 □ f3);
  l1 = (length bc1); l12 = (length (bc1@bc2));
  bc_mt_corresp bc2 f2 (sttp_of (f1 sttp0)) cG rT mxr (length bc2);
  length bc1 = length (mt_of (f1 sttp0));
  start_sttp_resp f2; start_sttp_resp f3⟧
⟹ bc_mt_corresp bc' f' sttp0 cG rT mxr l12"
```

⟨*proof*⟩

definition *contracting :: "(state_type ⇒ method_type × state_type) ⇒ bool"* **where**
"contracting f == (∀ ST LT.
let (ST', LT') = sttp_of (f (ST, LT))
in (length ST' ≤ length ST ∧ set ST' ⊆ set ST ∧
length LT' = length LT ∧ set LT' ⊆ set LT))"

lemma *set_drop_Suc [rule_format]: "∀xs. set (drop (Suc n) xs) ⊆ set (drop n xs)"*
⟨*proof*⟩

lemma *set_drop_le [rule_format,simp]: "∀n xs. n ≤ m ⟶ set (drop m xs) ⊆ set (drop n xs)"*
⟨*proof*⟩

declare *set_drop_subset [simp]*

lemma *contracting_popST [simp]: "contracting (popST n)"*
⟨*proof*⟩

lemma *contracting_nochangeST [simp]: "contracting nochangeST"*
⟨*proof*⟩

lemma *check_type_contracting: "⟦ check_type cG mxs mxr (OK (Some sttp)); contracting f⟧*
⟹ check_type cG mxs mxr (OK (Some (sttp_of (f sttp))))"
⟨*proof*⟩

lemma *bc_mt_corresp_comb_wt_instr: "*
⟦ bc_mt_corresp bc' f' sttp0 cG rT mxr l1;
bc' = (bc1@[inst]@bc3); f'= (f1 □ f2 □ f3);
l1 = (length bc1);
length bc1 = length (mt_of (f1 sttp0));
length (mt_of (f2 (sttp_of (f1 sttp0)))) = 1;
start_sttp_resp_cons f1; start_sttp_resp_cons f2; start_sttp_resp f3;

check_type cG (max_ssize (mt_sttp_flatten (f' sttp0))) mxr
(OK ((mt_sttp_flatten (f' sttp0)) ! (length bc1)))
⟶
wt_instr_altern inst cG rT
(mt_sttp_flatten (f' sttp0))
(max_ssize (mt_sttp_flatten (f' sttp0)))
mxr
(Suc (length bc'))
empty_et
(length bc1);

```
  contracting f2
 ⟧
⟹ bc_mt_corresp bc' f' sttp0 cG rT mxr (length (bc1@[inst]))"
  ⟨proof⟩
```

lemma *compTpExpr_LT_ST_rewr [simp]:*

```
  "⟦ wf_java_prog G; wf_java_mdecl G C ((mn, pTs), rT, (pns, lvars, blk, res));
     local_env G C (mn, pTs) pns lvars ⊢ ex :: T;
     is_inited_LT C pTs lvars LT⟧
  ⟹ sttp_of (compTpExpr (pns, lvars, blk, res) G ex (ST, LT)) = (T # ST, LT)"
  ⟨proof⟩
```

lemma *wt_method_compTpExpr_Exprs_corresp: "*

```
  ⟦ jmb = (pns,lvars,blk,res);
  wf_prog wf_java_mdecl G;
  wf_java_mdecl G C ((mn, pTs), rT, jmb);
  E = (local_env G C (mn, pTs) pns lvars)⟧
⟹
 (∀ ST LT T bc' f'.
  E ⊢ ex :: T ⟶
  (is_inited_LT C pTs lvars LT) ⟶
  bc' = (compExpr jmb ex) ⟶
  f' = (compTpExpr jmb G ex)
  ⟶ bc_mt_corresp bc' f' (ST, LT) (comp G) rT (length LT) (length bc'))
 ∧
 (∀ ST LT Ts.
  E ⊢ exs [::] Ts ⟶
  (is_inited_LT C pTs lvars LT)
  ⟶ bc_mt_corresp (compExprs jmb exs) (compTpExprs jmb G exs) (ST, LT) (comp G) rT (length
LT) (length (compExprs jmb exs)))"
  ⟨proof⟩
```

lemmas *wt_method_compTpExpr_corresp [rule_format (no_asm)] =*

```
  wt_method_compTpExpr_Exprs_corresp [THEN conjunct1]
```

lemma *wt_method_compTpStmt_corresp [rule_format (no_asm)]: "*

```
  ⟦ jmb = (pns,lvars,blk,res);
  wf_prog wf_java_mdecl G;
  wf_java_mdecl G C ((mn, pTs), rT, jmb);
  E = (local_env G C (mn, pTs) pns lvars)⟧
⟹
 (∀ ST LT T bc' f'.
  E ⊢ s√ ⟶
  (is_inited_LT C pTs lvars LT) ⟶
  bc' = (compStmt jmb s) ⟶
```

```
  f' = (compTpStmt jmb G s)
  ⟶ bc_mt_corresp bc' f' (ST, LT) (comp G) rT (length LT) (length bc'))"
```

⟨*proof*⟩

lemma *wt_method_compTpInit_corresp:*
```
"⟦ jmb = (pns,lvars,blk,res);
  wf_java_mdecl G C ((mn, pTs), rT, jmb); mxr = length LT;
  length LT = (length pns) + (length lvars) + 1;  vn ∈ set (map fst lvars);
  bc = (compInit jmb (vn,ty)); f = (compTpInit jmb (vn,ty));
  is_type G ty ⟧
  ⟹ bc_mt_corresp bc f (ST, LT) (comp G) rT mxr (length bc)"
```
⟨*proof*⟩

lemma *wt_method_compTpInitLvars_corresp_aux [rule_format (no_asm)]:*
```
"
  ∀ lvars_pre lvars0 ST LT.
  jmb = (pns,lvars0,blk,res) ∧
  lvars0 = (lvars_pre @ lvars) ∧
  length LT = (length pns) + (length lvars0) + 1 ∧
  wf_java_mdecl G C ((mn, pTs), rT, jmb)
 ⟶ bc_mt_corresp (compInitLvars jmb lvars) (compTpInitLvars jmb lvars) (ST, LT) (comp
G) rT
       (length LT) (length (compInitLvars jmb lvars))"
```
⟨*proof*⟩

lemma *wt_method_compTpInitLvars_corresp:*
```
"⟦ jmb = (pns,lvars,blk,res);
  wf_java_mdecl G C ((mn, pTs), rT, jmb);
  length LT = (length pns) + (length lvars) + 1; mxr = (length LT);
  bc = (compInitLvars jmb lvars); f= (compTpInitLvars jmb lvars) ⟧
  ⟹ bc_mt_corresp bc f (ST, LT) (comp G) rT mxr (length bc)"
```
⟨*proof*⟩

lemma *wt_method_comp_wo_return:*
```
"⟦ wf_prog wf_java_mdecl G;
  wf_java_mdecl G C ((mn, pTs), rT, jmb);
  bc = compInitLvars jmb lvars @ compStmt jmb blk @ compExpr jmb res;
  jmb = (pns,lvars,blk,res);
  f = (compTpInitLvars jmb lvars ▫ compTpStmt jmb G blk ▫ compTpExpr jmb G res);
  sttp = (start_ST, start_LT C pTs (length lvars));
  li = (length (inited_LT C pTs lvars))
  ⟧
⟹ bc_mt_corresp bc f sttp (comp G) rT  li (length bc)"
```
⟨*proof*⟩

lemma *check_type_start:*
"⟦ wf_mhead cG (mn, pTs) rT; is_class cG C⟧
⟹ check_type cG (length start_ST) (Suc (length pTs + mxl))
(OK (Some (start_ST, start_LT C pTs mxl)))"
⟨*proof*⟩

lemma *wt_method_comp_aux:*
"⟦ bc' = bc @ [Return]; f' = (f □ nochangeST);
bc_mt_corresp bc f sttp0 cG rT (1+length pTs+mxl) (length bc);
start_sttp_resp_cons f';
sttp0 = (start_ST, start_LT C pTs mxl);
mxs = max_ssize (mt_of (f' sttp0));
wf_mhead cG (mn, pTs) rT; is_class cG C;
sttp_of (f sttp0) = (T # ST, LT);

check_type cG mxs (1+length pTs+mxl) (OK (Some (T # ST, LT))) ⟶
wt_instr_altern Return cG rT (mt_of (f' sttp0)) mxs (1+length pTs+mxl)
(Suc (length bc)) empty_et (length bc)
⟧
⟹ wt_method_altern cG C pTs rT mxs mxl bc' empty_et (mt_of (f' sttp0))"
⟨*proof*⟩

lemma *wt_instr_Return: "⟦fst f ! pc = Some (T # ST, LT); (G ⊢ T ≼ rT); pc < max_pc;*
check_type (TranslComp.comp G) mxs mxr (OK (Some (T # ST, LT)))
⟧
⟹ wt_instr_altern Return (comp G) rT (mt_of f) mxs mxr max_pc empty_et pc"
⟨*proof*⟩

theorem *wt_method_comp: "*
⟦ wf_java_prog G; (C, D, fds, mths) ∈ set G; jmdcl ∈ set mths;
jmdcl = ((mn,pTs), rT, jmb);
mt = (compTpMethod G C jmdcl);
(mxs, mxl, bc, et) = mtd_mb (compMethod G C jmdcl) ⟧
⟹ wt_method (comp G) C pTs rT mxs mxl bc et mt"

⟨*proof*⟩

lemma *comp_set_ms: "(C, D, fs, cms)∈set (comp G)*
⟹ ∃ ms. (C, D, fs, ms) ∈set G ∧ cms = map (compMethod G C) ms"
⟨*proof*⟩

4.27.5 Main Theorem

theorem *wt_prog_comp: "wf_java_prog G ⟹ wt_jvm_prog (comp G) (compTp G)"*

⟨*proof*⟩

declare *split_paired_All [simp add]*
declare *split_paired_Ex [simp add]*

end
theory *MicroJava*
imports
"J/JTypeSafe"
"J/Example"
"J/JListExample"
"JVM/JVMListExample"
"JVM/JVMDefensive"
"BV/LBVJVM"
"BV/BVNoTypeError"
"BV/BVExample"
"Comp/CorrComp"
"Comp/CorrCompTp"
begin

end

Bibliography

[1] G. Klein and T. Nipkow. Verified lightweight bytecode verification. In S. Drossopoulou, S. Eisenbach, B. Jacobs, G. T. Leavens, P. Müller, and A. Poetzsch-Heffter, editors, *Formal Techniques for Java Programs*. Technical Report 269, 5/2000, Fernuniversität Hagen, 2000. ECOOP2000 Workshop proceedings available from http://www.informatik.fernuni-hagen.de/pi5/publications.html.

[2] G. Klein and T. Nipow. Verified lightweight bytecode verification. *Concurrency and Computation: Practice and Experience*, 13(13):1133–1151, 2001. Invited contribution to special issue on Formal Techniques for Java.

[3] T. Nipkow. Verified bytecode verifiers. In F. Honsell, editor, *Foundations of Software Science and Computation Structures (FOSSACS 2001)*, volume 2030, pages 347–363, 2001.

[4] T. Nipkow, D. v. Oheimb, and C. Pusch. μJava: Embedding a programming language in a theorem prover. In F. L. Bauer and R. Steinbrüggen, editors, *Foundations of Secure Computation*, volume 175 of *NATO Science Series F: Computer and Systems Sciences*, pages 117–144. IOS Press, 2000.

[5] D. von Oheimb. Axiomatic semantics for Java$^{\ell ight}$ in Isabelle/HOL. In S. Drossopoulou, S. Eisenbach, B. Jacobs, G. T. Leavens, P. Müller, and A. Poetzsch-Heffter, editors, *Formal Techniques for Java Programs*. Technical Report 269, 5/2000, Fernuniversität Hagen, 2000. ECOOP2000 Workshop proceedings available from http://www.informatik.fernuni-hagen.de/pi5/publications.html.