μJava
Java Virtual Machine and Bytecode Verifier
– Exceptions –

Gerwin Klein

November 28, 2002

Contents

1 Preface **5**

2 Program Structure and Declarations **7**
2.1 Some Auxiliary Definitions . . . 8
2.2 Java types . . . 10
2.3 Class Declarations and Programs . . . 11
2.4 System Classes . . . 12
2.5 Relations between Java Types . . . 13
2.6 Java Values . . . 17
2.7 Program State . . . 18
2.8 Expressions and Statements . . . 21
2.9 Well-formedness of Java programs . . . 22
2.10 Well-typedness Constraints . . . 33
2.11 Conformity Relations for Type Soundness Proof . . . 39

3 Java Virtual Machine **47**
3.1 State of the JVM . . . 48
3.2 Instructions of the JVM . . . 49
3.3 JVM Instruction Semantics . . . 50
3.4 Exception handling in the JVM . . . 53
3.5 Program Execution in the JVM . . . 54
3.6 Example for generating executable code from JVM semantics . . . 55
3.7 A Defensive JVM . . . 59

4 Bytecode Verifier **63**
4.1 Semilattices . . . 64
4.2 The Error Type . . . 71
4.3 Fixed Length Lists . . . 78
4.4 Typing and Dataflow Analysis Framework . . . 88
4.5 Products as Semilattices . . . 89
4.6 More on Semilattices . . . 92
4.7 Kildall's Algorithm . . . 96
4.8 Lifting the Typing Framework to err, app, and eff . . . 105
4.9 More about Options . . . 110
4.10 The Java Type System as Semilattice . . . 116
4.11 The JVM Type System as Semilattice . . . 122
4.12 Effect of Instructions on the State Type . . . 131
4.13 Monotonicity of eff and app . . . 140

4.14 The Bytecode Verifier . . . 149
4.15 The Typing Framework for the JVM . . . 152
4.16 Kildall for the JVM . . . 159
4.17 The Lightweight Bytecode Verifier . . . 165
4.18 Correctness of the LBV . . . 172
4.19 Completeness of the LBV . . . 176
4.20 LBV for the JVM . . . 184
4.21 BV Type Safety Invariant . . . 190
4.22 BV Type Safety Proof . . . 197
4.23 Welltyped Programs produce no Type Errors . . . 224
4.24 Example Welltypings . . . 233

Chapter 1

Preface

This document contains the automatically generated listings of the Isabelle sources for μJava with exception handling. The formalization is described in Chapter 3 of [1].
Figure 1.1 shows the dependencies between the Isabelle theories in the following sections.

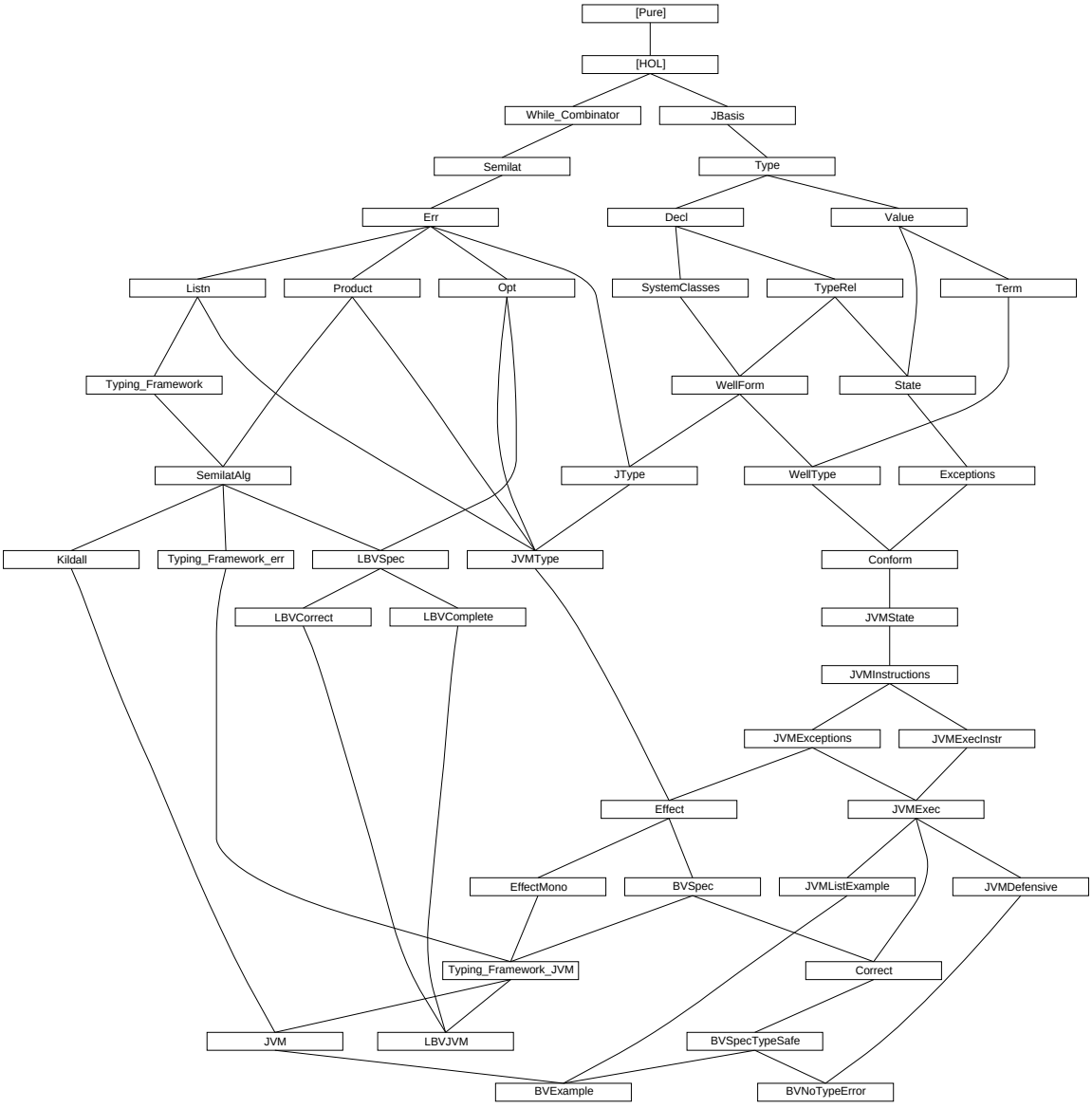

Figure 1.1: Theory Dependency Graph

Chapter 2

Program Structure and Declarations

2.1 Some Auxiliary Definitions

theory *JBasis = Main:*

lemmas *[simp] = Let_def*

2.1.1 unique

constdefs
```
  unique  :: "('a × 'b) list => bool"
  "unique  == distinct ∘ map fst"
```

lemma *fst_in_set_lemma [rule_format (no_asm)]:*
```
      "(x, y) : set xys --> x : fst ` set xys"
```
apply *(induct_tac "xys")*
apply *auto*
done

lemma *unique_Nil [simp]: "unique []"*
apply *(unfold unique_def)*
apply *(simp (no_asm))*
done

lemma *unique_Cons [simp]: "unique ((x,y)#l) = (unique l & (!y. (x,y) ~: set l))"*
apply *(unfold unique_def)*
apply *(auto dest: fst_in_set_lemma)*
done

lemma *unique_append [rule_format (no_asm)]: "unique l' ==> unique l -->*
(!(x,y):set l. !(x',y'):set l'. x' ~= x) --> unique (l @ l')"
apply *(induct_tac "l")*
apply *(auto dest: fst_in_set_lemma)*
done

lemma *unique_map_inj [rule_format (no_asm)]:*
```
  "unique l --> inj f --> unique (map (%(k,x). (f k, g k x)) l)"
```
apply *(induct_tac "l")*
apply *(auto dest: fst_in_set_lemma simp add: inj_eq)*
done

2.1.2 More about Maps

lemma *map_of_SomeI [rule_format (no_asm)]:*
```
  "unique l --> (k, x) : set l --> map_of l k = Some x"
```
apply *(induct_tac "l")*
apply *auto*
done

lemma *Ball_set_table_:*
```
  "(∀(x,y)∈set l. P x y) --> (∀x. ∀y. map_of l x = Some y --> P x y)"
```
apply(*induct_tac "l"*)
apply(*simp_all (no_asm)*)
apply *safe*
apply *auto*

done
lemmas *Ball_set_table = Ball_set_table_ [THEN mp]*

lemma *table_of_remap_SomeD [rule_format (no_asm)]:*
"map_of (map (λ((k,k'),x). (k,(k',x))) t) k = Some (k',x) -->
map_of t (k, k') = Some x"
apply *(induct_tac "t")*
apply *auto*
done

end

2.2 Java types

theory *Type = JBasis:*

typedecl *cnam*

— exceptions
datatype
  xcpt
  = NullPointer
  | ClassCast
  | OutOfMemory

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

typedecl *vnam* — variable or field name
typedecl *mname* — method name

— 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

syntax
  NT :: "ty"
  Class :: "cname => ty"

translations
  "NT" == "RefT NullT"
  "Class C" == "RefT (ClassT C)"

end

2.3 Class Declarations and Programs

theory *Decl = Type:*

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

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

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

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

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

  'c prog = "'c cdecl list" — program

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

constdefs
```
  class :: "'c prog => (cname ⇀ 'c class)"
  "class ≡ map_of"

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

lemma *finite_is_class: "finite {C. is_class G C}"*
apply *(unfold is_class_def class_def)*
apply *(fold dom_def)*
apply *(rule finite_dom_map_of)*
done

consts
```
  is_type :: "'c prog => ty    => bool"
```
primrec
```
  "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 System Classes

theory *SystemClasses = Decl:*

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

constdefs

```
  ObjectC :: "'c cdecl"
  "ObjectC ≡ (Object, (arbitrary,[],[]))"

  NullPointerC :: "'c cdecl"
  "NullPointerC ≡ (Xcpt NullPointer, (Object,[],[]))"

  ClassCastC :: "'c cdecl"
  "ClassCastC ≡ (Xcpt ClassCast, (Object,[],[]))"

  OutOfMemoryC :: "'c cdecl"
  "OutOfMemoryC ≡ (Xcpt OutOfMemory, (Object,[],[]))"

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

end

2.5 Relations between Java Types

theory *TypeRel = Decl:*

consts
```
  subcls1 :: "'c prog => (cname × cname) set"  — subclass
  widen   :: "'c prog => (ty    × ty   ) set"  — widening
  cast    :: "'c prog => (cname × cname) set"  — casting
```

syntax *(xsymbols)*
```
  subcls1 :: "'c prog => [cname, cname] => bool" ("_ ⊢ _ ≺C1 _" [71,71,71] 70)
  subcls  :: "'c prog => [cname, cname] => bool" ("_ ⊢ _ ≼C _"  [71,71,71] 70)
  widen   :: "'c prog => [ty   , ty   ] => bool" ("_ ⊢ _ ≼ _"   [71,71,71] 70)
  cast    :: "'c prog => [cname, cname] => bool" ("_ ⊢ _ ≼? _"  [71,71,71] 70)
```

syntax
```
  subcls1 :: "'c prog => [cname, cname] => bool" ("_ |- _ <=C1 _" [71,71,71] 70)
  subcls  :: "'c prog => [cname, cname] => bool" ("_ |- _ <=C _"  [71,71,71] 70)
  widen   :: "'c prog => [ty   , ty   ] => bool" ("_ |- _ <= _"   [71,71,71] 70)
  cast    :: "'c prog => [cname, cname] => bool" ("_ |- _ <=? _"  [71,71,71] 70)
```

translations
```
  "G⊢C ≺C1 D" == "(C,D) ∈ subcls1 G"
  "G⊢C ≼C  D" == "(C,D) ∈ (subcls1 G)^*"
  "G⊢S ≼   T" == "(S,T) ∈ widen   G"
  "G⊢C ≼?  D" == "(C,D) ∈ cast    G"
```

— direct subclass, cf. 8.1.3
inductive *"subcls1 G"* **intros**
```
  subcls1I: "⟦class G C = Some (D,rest); C ≠ Object⟧ ⟹ G⊢C≺C1D"
```

lemma *subcls1D:*
```
  "G⊢C≺C1D ⟹ C ≠ Object ∧ (∃fs ms. class G C = Some (D,fs,ms))"
```
apply *(erule subcls1.elims)*
apply *auto*
done

lemma *subcls1_def2:*
```
"subcls1 G = (ΣC∈{C. is_class G C} . {D. C≠Object ∧ fst (the (class G C))=D})"
```
  by *(auto simp add: is_class_def dest: subcls1D intro: subcls1I)*

lemma *finite_subcls1: "finite (subcls1 G)"*
apply*(subst subcls1_def2)*
apply*(rule finite_SigmaI [OF finite_is_class])*
apply*(rule_tac B = "{fst (the (class G C))}"* **in** *finite_subset)*
apply *auto*
done

lemma *subcls_is_class: "(C,D) ∈ (subcls1 G)^+ ==> is_class G C"*
apply *(unfold is_class_def)*
apply*(erule trancl_trans_induct)*
apply *(auto dest!: subcls1D)*
done

```
lemma subcls_is_class2 [rule_format (no_asm)]:
  "G⊢C⪯C D ⟹ is_class G D ⟶ is_class G C"
apply (unfold is_class_def)
apply (erule rtrancl_induct)
apply  (drule_tac [2] subcls1D)
apply  auto
done

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

lemma class_rec_lemma: "wf ((subcls1 G)^-1) ⟹ class G C = Some (D,fs,ms) ⟹
 class_rec G C t f = f C fs ms (if C=Object then t else class_rec G D t f)"
  by (simp add: class_rec_def wfrec cut_apply [OF converseI [OF subcls1I]])

consts

  method :: "'c prog × cname => ( sig   ⇝ cname × ty × 'c)"
  field  :: "'c prog × cname => ( vname ⇝ cname × ty     )"
  fields :: "'c prog × cname => ((vname × cname) × ty) list"
```

— methods of a class, with inheritance, overriding and hiding, cf. 8.4.6

```
defs method_def: "method ≡ λ(G,C). class_rec G C empty (λC fs ms ts.
                           ts ++ map_of (map (λ(s,m). (s,(C,m))) ms))"

lemma method_rec_lemma: "[|class G C = Some (D,fs,ms); wf ((subcls1 G)^-1)|] ==>
  method (G,C) = (if C = Object then empty else method (G,D)) ++
  map_of (map (λ(s,m). (s,(C,m))) ms)"
apply (unfold method_def)
apply (simp split del: split_if)
apply (erule (1) class_rec_lemma [THEN trans])
apply auto
done
```

— list of fields of a class, including inherited and hidden ones

```
defs fields_def: "fields ≡ λ(G,C). class_rec G C []    (λC fs ms ts.
                           map (λ(fn,ft). ((fn,C),ft)) fs @ ts)"

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))"
apply (unfold fields_def)
apply (simp split del: split_if)
apply (erule (1) class_rec_lemma [THEN trans])
apply auto
done
```

defs *field_def: "field == map_of o (map (λ((fn,fd),ft). (fn,(fd,ft)))) o fields"*

lemma *field_fields:*
"field (G,C) fn = Some (fd, fT) ⟹ map_of (fields (G,C)) (fn, fd) = Some fT"
apply *(unfold field_def)*
apply *(rule table_of_remap_SomeD)*
apply *simp*
done

— widening, viz. method invocation conversion,cf. 5.3 i.e. sort of syntactic subtyping
inductive *"widen G"* **intros**
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"

— casting conversion, cf. 5.5 / 5.1.5
— left out casts on primitve types
inductive *"cast G"* **intros**
widen: "G⊢C≼C D ==> G⊢C ≼? D"
subcls: "G⊢D≼C C ==> G⊢C ≼? D"

lemma *widen_PrimT_RefT [iff]: "(G⊢PrimT pT≼RefT rT) = False"*
apply *(rule iffI)*
apply *(erule widen.elims)*
apply *auto*
done

lemma *widen_RefT: "G⊢RefT R≼T ==> ∃t. T=RefT t"*
apply *(ind_cases "G⊢S≼T")*
apply *auto*
done

lemma *widen_RefT2: "G⊢S≼RefT R ==> ∃t. S=RefT t"*
apply *(ind_cases "G⊢S≼T")*
apply *auto*
done

lemma *widen_Class: "G⊢Class C≼T ==> ∃D. T=Class D"*
apply *(ind_cases "G⊢S≼T")*
apply *auto*
done

lemma *widen_Class_NullT [iff]: "(G⊢Class C≼NT) = False"*
apply *(rule iffI)*
apply *(ind_cases "G⊢S≼T")*
apply *auto*
done

lemma *widen_Class_Class [iff]: "(G⊢Class C≼ Class D) = (G⊢C≼C D)"*
apply *(rule iffI)*
apply *(ind_cases "G⊢S≼T")*
apply *(auto elim: widen.subcls)*

done

theorem *widen_trans[trans]:* "⟦*G*⊢*S*≼*U*; *G*⊢*U*≼*T*⟧ ⟹ *G*⊢*S*≼*T*"
proof -
  assume "*G*⊢*S*≼*U*" **thus** "⋀*T*. *G*⊢*U*≼*T* ⟹ *G*⊢*S*≼*T*"
  proof *induct*
    case *(refl T T')* **thus** "*G*⊢*T*≼*T'*" .
  next
    case *(subcls C D T)*
    then obtain *E* **where** "*T* = *Class E*" **by** *(blast dest: widen_Class)*
    with *subcls* **show** "*G*⊢*Class C*≼*T*" **by** *(auto elim: rtrancl_trans)*
  next
    case *(null R RT)*
    then obtain *rt* **where** "*RT* = *RefT rt*" **by** *(blast dest: widen_RefT)*
    thus "*G*⊢*NT*≼*RT*" **by** *auto*
  qed
qed

end

2.6 Java Values

theory *Value = Type:*

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

consts
  the_Bool :: "val => bool"
  the_Intg :: "val => int"
  the_Addr :: "val => loc"

primrec
  "the_Bool (Bool b) = b"

primrec
  "the_Intg (Intg i) = i"

primrec
  "the_Addr (Addr a) = a"

consts
  defpval :: "prim_ty => val" — default value for primitive types
  default_val :: "ty => val" — default value for all types

primrec
  "defpval Void   = Unit"
  "defpval Boolean = Bool False"
  "defpval Integer = Intg 0"

primrec
  "default_val (PrimT pt) = defpval pt"
  "default_val (RefT r) = Null"

end

2.7 Program State

theory *State = TypeRel + Value:*

types
fields_ = "(vname × cname ⇝ val)" — field name, defining class, value

obj = "cname × fields_" — class instance with class name and fields

constdefs
```
  obj_ty  :: "obj => ty"
 "obj_ty obj  == Class (fst obj)"

  init_vars :: "('a × ty) list => ('a ⇝ val)"
 "init_vars == map_of o map (λ(n,T). (n,default_val T))"
```

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

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

syntax
```
  heap    :: "state => aheap"
  locals  :: "state => locals"
  Norm    :: "state => xstate"
  abrupt  :: "xstate ⇒ val option"
  store   :: "xstate ⇒ state"
  lookup_obj   :: "state ⇒ val ⇒ obj"
```

translations
```
  "heap"   => "fst"
  "locals" => "snd"
  "Norm s" == "(None,s)"
  "abrupt"     => "fst"
  "store"      => "snd"
 "lookup_obj s a'"  == "the (heap s (the_Addr a'))"
```

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

  new_Addr  :: "aheap => loc × val option"
  "new_Addr h ≡ SOME (a,x). (h a = None ∧  x = None) |  x = Some (Addr (XcptRef OutOfMemory))"

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

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

  cast_ok :: "'c prog => cname => aheap => val => bool"
```

```
 "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"
apply (unfold obj_ty_def)
apply (simp (no_asm))
done


lemma new_AddrD: "new_Addr hp = (ref, xcp) ⟹
  hp ref = None ∧ xcp = None ∨ xcp = Some (Addr (XcptRef OutOfMemory))"
apply (drule sym)
apply (unfold new_Addr_def)
apply(simp add: Pair_fst_snd_eq Eps_split)
apply(rule someI)
apply(rule disjI2)
apply(rule_tac "r" = "snd (?a,Some (Addr (XcptRef OutOfMemory)))" in trans)
apply auto
done

lemma raise_if_True [simp]: "raise_if True x y ≠ None"
apply (unfold raise_if_def)
apply auto
done

lemma raise_if_False [simp]: "raise_if False x y = y"
apply (unfold raise_if_def)
apply auto
done

lemma raise_if_Some [simp]: "raise_if c x (Some y) ≠ None"
apply (unfold raise_if_def)
apply auto
done

lemma raise_if_Some2 [simp]:
  "raise_if c z (if x = None then Some y else x) ≠ None"
apply (unfold raise_if_def)
apply(induct_tac "x")
apply auto
done

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"
apply (unfold raise_if_def)
apply auto
done

lemma raise_if_NoneD [rule_format (no_asm)]:
  "raise_if c x y = None --> ¬ c ∧  y = None"
apply (unfold raise_if_def)
apply auto
done

lemma np_NoneD [rule_format (no_asm)]:
```

```
  "np a' x' = None --> x' = None ∧  a' ≠ Null"
apply (unfold np_def raise_if_def)
apply auto
done

lemma np_None [rule_format (no_asm), simp]: "a' ≠ Null --> np a' x' = x'"
apply (unfold np_def raise_if_def)
apply auto
done

lemma np_Some [simp]: "np a' (Some xc) = Some xc"
apply (unfold np_def raise_if_def)
apply auto
done

lemma np_Null [simp]: "np Null None = Some (Addr (XcptRef NullPointer))"
apply (unfold np_def raise_if_def)
apply auto
done

lemma np_Addr [simp]: "np (Addr a) None = None"
apply (unfold np_def raise_if_def)
apply auto
done

lemma np_raise_if [simp]: "(np Null (raise_if c xc None)) =
  Some (Addr (XcptRef (if c then  xc else NullPointer)))"
apply (unfold raise_if_def)
apply (simp (no_asm))
done

end
```

2.8 Expressions and Statements

theory *Term = Value:*

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 *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

2.9 Well-formedness of Java programs

theory *WellForm = TypeRel + SystemClasses:*

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

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

constdefs

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

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

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

 wf_cdecl :: "'c wf_mb => 'c prog => 'c cdecl => bool"
"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'))"

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

 wf_prog :: "'c wf_mb => 'c prog => bool"
"wf_prog wf_mb G ==
   let cs = set G in wf_syscls G ∧ (∀c∈cs. wf_cdecl wf_mb G c) ∧ unique G"
```

lemma *class_wf:*

```
 "[|class G C = Some c; wf_prog wf_mb G|] ==> wf_cdecl wf_mb G (C,c)"
```

apply *(unfold wf_prog_def class_def)*
apply *(simp)*
apply *(fast dest: map_of_SomeD)*
done

lemma *class_Object [simp]:*

```
  "wf_prog wf_mb G ==> ∃X fs ms. class G Object = Some (X,fs,ms)"
```

apply *(unfold wf_prog_def wf_syscls_def class_def)*
apply *(auto simp: map_of_SomeI)*
done

lemma *is_class_Object [simp]: "wf_prog wf_mb G ==> is_class G Object"*
apply *(unfold is_class_def)*
apply *(simp (no_asm_simp))*
done

lemma *is_class_xcpt [simp]: "wf_prog wf_mb G ⟹ is_class G (Xcpt x)"*
  apply *(simp add: wf_prog_def wf_syscls_def)*
  apply *(simp add: is_class_def class_def)*
  apply *clarify*
  apply *(erule_tac x = x* **in** *allE)*
  apply *clarify*
  apply *(auto intro!: map_of_SomeI)*
  done

lemma *subcls1_wfD: "[|G⊢C≺C1D; wf_prog wf_mb G|] ==> D ≠ C ∧ ¬(D,C)∈(subcls1 G)^+"*
apply(*frule r_into_trancl)*
apply(*drule subcls1D)*
apply(*clarify)*
apply(*drule (1) class_wf)*
apply(*unfold wf_cdecl_def)*
apply(*force simp add: reflcl_trancl [THEN sym] simp del: reflcl_trancl)*
done

lemma *wf_cdecl_supD:*
"!!r. ⟦wf_cdecl wf_mb G (C,D,r); C ≠ Object⟧ ⟹ is_class G D"
apply *(unfold wf_cdecl_def)*
apply *(auto split add: option.split_asm)*
done

lemma *subcls_asym: "[|wf_prog wf_mb G; (C,D)∈(subcls1 G)^+|] ==> ¬(D,C)∈(subcls1 G)^+"*
apply(*erule tranclE)*
apply(*fast dest!: subcls1_wfD)*
apply(*fast dest!: subcls1_wfD intro: trancl_trans)*
done

lemma *subcls_irrefl: "[|wf_prog wf_mb G; (C,D)∈(subcls1 G)^+|] ==> C ≠ D"*
apply *(erule trancl_trans_induct)*
apply *(auto dest: subcls1_wfD subcls_asym)*
done

lemma *acyclic_subcls1: "wf_prog wf_mb G ==> acyclic (subcls1 G)"*
apply *(unfold acyclic_def)*
apply *(fast dest: subcls_irrefl)*
done

lemma *wf_subcls1: "wf_prog wf_mb G ==> wf ((subcls1 G)^-1)"*
apply *(rule finite_acyclic_wf)*
apply *(subst finite_converse)*
apply *(rule finite_subcls1)*
apply *(subst acyclic_converse)*

```
apply (erule acyclic_subcls1)
done

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 -
  assume p: "PROP ?P"
  assume ?A thus ?thesis apply -
apply(drule wf_subcls1)
apply(drule wf_trancl)
apply(simp only: trancl_converse)
apply(erule_tac a = C in wf_induct)
apply(rule p)
apply(auto)
done
qed

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 -
  assume p: "PROP ?P"
  assume ?A ?B ?C thus ?thesis apply -
apply(unfold is_class_def)
apply( rule impE)
prefer 2
apply(   assumption)
prefer 2
apply(  assumption)
apply( erule thin_rl)
apply( rule subcls_induct)
apply(  assumption)
apply( rule impI)
apply( case_tac "C = Object")
apply(  fast)
apply safe
apply( frule (1) class_wf)
apply( frule (1) wf_cdecl_supD)

apply( subgoal_tac "G⊢C≺C1a")
apply( erule_tac [2] subcls1I)
apply(  rule p)
apply (unfold is_class_def)
apply auto
done
qed

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); wf_prog wf_mb G⟧
⟹ field (G, C) =
   (if C = Object then empty else field (G, D)) ++
   map_of (map (λ(s, f). (s, C, f)) fs)"
apply (simp only: field_def)
apply (frule fields_rec, assumption)
apply (rule HOL.trans)
apply (simp add: o_def)
apply (simp (no_asm_use)
  add: split_beta split_def o_def map_compose [THEN sym] del: map_compose)
done

lemma method_Object [simp]:
  "method (G, Object) sig = Some (D, mh, code) ⟹ wf_prog wf_mb G ⟹ D = Object"
  apply (frule class_Object, clarify)
  apply (drule method_rec, assumption)
  apply (auto dest: map_of_SomeD)
  done


lemma fields_Object [simp]: "⟦ ((vn, C), T) ∈ set (fields (G, Object)); wf_prog wf_mb
G ⟧
  ⟹ C = Object"
apply (frule class_Object)
apply clarify
apply (subgoal_tac "fields (G, Object) = map (λ(fn,ft). ((fn,Object),ft)) fs")
apply (simp add: image_iff split_beta)
apply auto
apply (rule trans)
apply (rule fields_rec, assumption+)
apply simp
done

lemma subcls_C_Object: "[|is_class G C; wf_prog wf_mb G|] ==> G⊢C≼C Object"
apply(erule subcls1_induct)
apply(  assumption)
apply( fast)
apply(auto dest!: wf_cdecl_supD)
apply(erule (1) converse_rtrancl_into_rtrancl)
done

lemma is_type_rTI: "wf_mhead G sig rT ==> is_type G rT"
apply (unfold wf_mhead_def)
apply auto
done

lemma widen_fields_defpl': "[|is_class G C; wf_prog wf_mb G|] ==>
  ∀((fn,fd),fT)∈set (fields (G,C)). G⊢C≼C fd"
apply( erule subcls1_induct)
apply(   assumption)
apply(  frule class_Object)
apply(  clarify)
apply(  frule fields_rec, assumption)
```

```
apply(  fastsimp)
apply( tactic "safe_tac HOL_cs")
apply( subst fields_rec)
apply(   assumption)
apply(  assumption)
apply( simp (no_asm) split del: split_if)
apply( rule ballI)
apply( simp (no_asm_simp) only: split_tupled_all)
apply( simp (no_asm))
apply( erule UnE)
apply(  force)
apply( erule r_into_rtrancl [THEN rtrancl_trans])
apply auto
done

lemma widen_fields_defpl:
  "[|((fn,fd),fT) ∈ set (fields (G,C)); wf_prog wf_mb G; is_class G C|] ==>
  G⊢C≼C fd"
apply( drule (1) widen_fields_defpl')
apply (fast)
done

lemma unique_fields:
  "[|is_class G C; wf_prog wf_mb G|] ==> unique (fields (G,C))"
apply( erule subcls1_induct)
apply(   assumption)
apply(  frule class_Object)
apply(  clarify)
apply(  frule fields_rec, assumption)
apply(  drule class_wf, assumption)
apply(  simp add: wf_cdecl_def)
apply(  rule unique_map_inj)
apply(   simp)
apply(  rule inj_onI)
apply(  simp)
apply( safe dest!: wf_cdecl_supD)
apply( drule subcls1_wfD)
apply(  assumption)
apply( subst fields_rec)
apply   auto
apply( rotate_tac -1)
apply( frule class_wf)
apply  auto
apply( simp add: wf_cdecl_def)
apply( erule unique_append)
apply(  rule unique_map_inj)
apply(   clarsimp)
apply  (rule inj_onI)
apply(  simp)
apply(auto dest!: widen_fields_defpl)
done

lemma fields_mono_lemma [rule_format (no_asm)]:
  "[|wf_prog wf_mb G; (C',C)∈(subcls1 G)^*|] ==>
```

```
  x ∈ set (fields (G,C)) --> x ∈ set (fields (G,C'))"
apply(erule converse_rtrancl_induct)
apply( safe dest!: subcls1D)
apply(subst fields_rec)
apply(  auto)
done

lemma fields_mono:
"⟦map_of (fields (G,C)) fn = Some f; G⊢D≼C C; is_class G D; wf_prog wf_mb G⟧
  ⟹ map_of (fields (G,D)) fn = Some f"
apply (rule map_of_SomeI)
apply  (erule (1) unique_fields)
apply (erule (1) fields_mono_lemma)
apply (erule map_of_SomeD)
done

lemma widen_cfs_fields:
"[|field (G,C) fn = Some (fd, fT); G⊢D≼C C; wf_prog wf_mb G|]==>
  map_of (fields (G,D)) (fn, fd) = Some fT"
apply (drule field_fields)
apply (drule rtranclD)
apply safe
apply (frule subcls_is_class)
apply (drule trancl_into_rtrancl)
apply (fast dest: fields_mono)
done

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))"
apply( erule subcls1_induct)
apply(   assumption)
apply(  clarify)
apply(  frule class_Object)
apply(  clarify)
apply(  frule method_rec, assumption)
apply(  drule class_wf, assumption)
apply(  simp add: wf_cdecl_def)
apply(  drule map_of_SomeD)
apply(  subgoal_tac "md = Object")
apply(   fastsimp)
apply(  fastsimp)
apply( clarify)
apply( frule_tac C = C in method_rec)
apply(  assumption)
apply( rotate_tac -1)
apply( simp)
apply( drule override_SomeD)
apply( erule disjE)
apply(  erule_tac V = "?P --> ?Q" in thin_rl)
apply (frule map_of_SomeD)
apply (clarsimp simp add: wf_cdecl_def)
apply( clarify)
```

```
apply( rule rtrancl_trans)
prefer 2
apply(  assumption)
apply( rule r_into_rtrancl)
apply( fast intro: subcls1I)
done


lemma wf_prog_wf_mhead: "⟦ wf_prog wf_mb G; (C, D, fds, mths) ∈ set G;
  ((mn, pTs), rT, jmb) ∈ set mths ⟧
  ⟹ wf_mhead G (mn, pTs) rT"
apply (simp add: wf_prog_def wf_cdecl_def)
apply (erule conjE)+
apply (drule bspec, assumption)
apply simp
apply (erule conjE)+
apply (drule bspec, assumption)
apply (simp add: wf_mdecl_def)
done

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⊢rT'⪯rT)"
apply( drule rtranclD)
apply( erule disjE)
apply(  fast)
apply( erule conjE)
apply( erule trancl_trans_induct)
prefer 2
apply(  clarify)
apply(  drule spec, drule spec, drule spec, erule (1) impE)
apply(  fast elim: widen_trans)
apply( clarify)
apply( drule subcls1D)
apply( clarify)
apply( subst method_rec)
apply(  assumption)
apply( unfold override_def)
apply( simp (no_asm_simp) del: split_paired_Ex)
apply( case_tac "∃z. map_of(map (λ(s,m). (s, ?C, m)) ms) sig = Some z")
apply(  erule exE)
apply(  rotate_tac -1, frule ssubst, erule_tac [2] asm_rl)
prefer 2
apply(  rotate_tac -1, frule ssubst, erule_tac [2] asm_rl)
apply(  tactic "asm_full_simp_tac (HOL_ss addsimps [not_None_eq RS sym]) 1")
apply(  simp_all (no_asm_simp) del: split_paired_Ex)
apply( drule (1) class_wf)
apply( simp (no_asm_simp) only: split_tupled_all)
apply( unfold wf_cdecl_def)
apply( drule map_of_SomeD)
apply auto
done
```

```
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"
apply(auto dest: subcls_widen_methd method_wf_mdecl
           simp add: wf_mdecl_def wf_mhead_def split_def)
done

lemma method_in_md [rule_format (no_asm)]:
  "wf_prog wf_mb 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)"
apply (erule (1) subcls1_induct)
 apply clarify
 apply (frule method_Object, assumption)
 apply hypsubst
 apply simp
apply (erule conjE)
apply (subst method_rec)
  apply (assumption)
 apply (assumption)
apply (clarify)
apply (erule_tac "x" = "Da" in allE)
apply (clarsimp)
 apply (simp add: map_of_map)
 apply (clarify)
 apply (subst method_rec)
   apply (assumption)
  apply (assumption)
 apply (simp add: override_def map_of_map split add: option.split)
done


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))))"
apply (erule (1) subcls1_induct)

apply clarify
apply (frule fields_Object, assumption+)
apply (simp only: is_class_Object) apply simp

apply clarify
apply (frule fields_rec)
apply assumption

apply (case_tac "Da=C")
apply blast

apply (subgoal_tac "((vn, Da), T) ∈ set (fields (G, D))") apply blast
apply (erule thin_rl)
apply (rotate_tac 1)
apply (erule thin_rl, erule thin_rl, erule thin_rl,
      erule thin_rl, erule thin_rl, erule thin_rl)
apply auto
```

done

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))"*
apply *(erule (1) subcls1_induct)*

apply *clarify*
apply *(frule field_fields)*
apply *(drule map_of_SomeD)*
apply *(drule fields_Object, assumption+)*
apply *simp*

apply *clarify*
apply *(subgoal_tac "((field (G, D)) ++ map_of (map (λ(s, f). (s, C, f)) fs)) vn = Some (Da, T)")*
apply *(simp (no_asm_use) only: override_Some_iff)*
apply *(erule disjE)*
apply *(simp (no_asm_use) add: map_of_map)* **apply** *blast*
apply *blast*
apply *(rule trans [THEN sym], rule sym, assumption)*
apply *(rule_tac x=vn* **in** *fun_cong)*
apply *(rule trans, rule field_rec, assumption+)*
apply *(simp (no_asm_use))* **apply** *blast*
done

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"*
apply(*drule subcls_widen_methd)*
apply *auto*
done

lemma *widen_field: "⟦ (field (G,C) fn) = Some (fd, fT); wf_prog wf_mb G; is_class G C ⟧*
* ⟹ G⊢C⪯C fd"*
apply *(rule widen_fields_defpl)*
apply *(simp add: field_def)*
apply *(rule map_of_SomeD)*
apply *(rule table_of_remap_SomeD)*
apply *assumption+*
done

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)"*
apply(*drule (2) widen_methd)*
apply(*clarify)*
apply(*frule subcls_is_class2)*
apply *(unfold is_class_def)*
apply *(simp (no_asm_simp))*
apply(*drule method_wf_mdecl)*

```
apply( unfold wf_mdecl_def)
apply( unfold is_class_def)
apply auto
done


lemma fields_is_type_lemma [rule_format (no_asm)]:
  "[|is_class G C; wf_prog wf_mb G|] ==>
  ∀f∈set (fields (G,C)). is_type G (snd f)"
apply( erule (1) subcls1_induct)
apply(  frule class_Object)
apply(  clarify)
apply(  frule fields_rec, assumption)
apply(  drule class_wf, assumption)
apply(  simp add: wf_cdecl_def wf_fdecl_def)
apply(  fastsimp)
apply( subst fields_rec)
apply(   fast)
apply(  assumption)
apply( clarsimp)
apply( safe)
prefer 2
apply(  force)
apply( drule (1) class_wf)
apply( unfold wf_cdecl_def)
apply( clarsimp)
apply( drule (1) bspec)
apply( unfold wf_fdecl_def)
apply auto
done

lemma fields_is_type:
  "[|map_of (fields (G,C)) fn = Some f; wf_prog wf_mb G; is_class G C|] ==>
  is_type G f"
apply(drule map_of_SomeD)
apply(drule (2) fields_is_type_lemma)
apply(auto)
done

lemma methd:
  "[| wf_prog wf_mb 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 -
  assume wf: "wf_prog wf_mb G" and C:  "(C,S,fs,mdecls) ∈ set G" and
         m: "(sig,rT,code) ∈ set mdecls"
  moreover
  from wf C have "class G C = Some (S,fs,mdecls)"
    by (auto simp add: wf_prog_def class_def is_class_def intro: map_of_SomeI)
  moreover
  from wf C
  have "unique mdecls" by (unfold wf_prog_def wf_cdecl_def) auto
  hence "unique (map (λ(s,m). (s,C,m)) mdecls)" by (induct mdecls, auto)
  with m
  have "map_of (map (λ(s,m). (s,C,m)) mdecls) sig = Some (C,rT,code)"
```

```
    by (force intro: map_of_SomeI)
  ultimately
  show ?thesis by (auto simp add: is_class_def dest: method_rec)
qed


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"
  apply (simp add: wf_prog_def)
  apply auto
  apply (simp add: wf_cdecl_def wf_mdecl_def)
  apply safe
  apply (drule bspec, assumption) apply simp
  apply (drule bspec, assumption) apply simp
  apply (drule bspec, assumption) apply simp
  apply clarify apply (drule bspec, assumption) apply simp
  apply (drule bspec, assumption) apply simp
  apply (drule bspec, assumption) apply simp
  apply (drule bspec, assumption) apply simp
  apply (drule bspec, assumption) apply simp
  apply (drule bspec, assumption) apply simp
  apply clarify apply (drule bspec, assumption)+ apply simp
  done


lemma fst_mono: "A ⊆ B ⟹ fst ` A ⊆ fst ` B" by fast

lemma wf_syscls:
  "set SystemClasses ⊆ set G ⟹ wf_syscls G"
  apply (drule fst_mono)
  apply (simp add: SystemClasses_def wf_syscls_def)
  apply (simp add: ObjectC_def)
  apply (rule allI, case_tac x)
  apply (auto simp add: NullPointerC_def ClassCastC_def OutOfMemoryC_def)
  done

end
```

2.10 Well-typedness Constraints

theory *WellType = Term + WellForm:*

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:

types
```
  lenv   = "vname ⇀ ty"
  'c env = "'c prog × lenv"
```

syntax
```
  prg    :: "'c env => 'c prog"
  localT :: "'c env => (vname ⇀ ty)"
```

translations
```
  "prg"    => "fst"
  "localT" => "snd"
```

consts
```
  more_spec :: "'c prog => (ty × 'x) × ty list =>
                (ty × 'x) × ty list => bool"
  appl_methds :: "'c prog =>  cname => sig => ((ty × ty) × ty list) set"
  max_spec :: "'c prog =>  cname => sig => ((ty × ty) × ty list) set"
```

defs
```
  more_spec_def: "more_spec G == λ((d,h),pTs). λ((d',h'),pTs'). G⊢d⪯d' ∧
                                  list_all2 (λT T'. G⊢T⪯T') pTs pTs'"

  — applicable methods, cf. 15.11.2.1
  appl_methds_def: "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'}"

  — maximally specific methods, cf. 15.11.2.2
  max_spec_def: "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"
```
apply *(unfold max_spec_def)*
apply *(fast)*
done

```
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'"
apply (unfold appl_methds_def)
apply (fast)
done

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


consts
  typeof :: "(loc => ty option) => val => ty option"

primrec
  "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)"
apply (rule val.induct)
apply     auto
done

lemma typeof_empty_is_type [rule_format (no_asm)]:
  "typeof (λa. None) v = Some T ⟶ is_type G T"
apply (rule val.induct)
apply     auto
done

lemma typeof_default_val: "∃T. (typeof dt (default_val ty) = Some T) ∧ G⊢ T ⪯ ty"
apply (case_tac ty)
apply (case_tac prim_ty)
apply auto
done

types
  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

consts
  ty_expr :: "(java_mb env × expr      × ty     ) set"
  ty_exprs:: "(java_mb env × expr list × ty list) set"
  wt_stmt :: "(java_mb env × stmt               ) set"

syntax (xsymbols)
  ty_expr :: "java_mb env => [expr     , ty     ] => bool" ("_ ⊢ _ :: _"   [51,51,51]50)
  ty_exprs:: "java_mb env => [expr list, ty list] => bool" ("_ ⊢ _ [::] _" [51,51,51]50)
```

```
  wt_stmt :: "java_mb env =>  stmt                => bool" ("_ ⊢ _ √"      [51,51   ]50)
```

syntax

```
  ty_expr :: "java_mb env => [expr     , ty     ] => bool" ("_ |- _ :: _"   [51,51,51]50)
  ty_exprs:: "java_mb env => [expr list, ty list] => bool" ("_ |- _ [::] _" [51,51,51]50)
  wt_stmt :: "java_mb env =>  stmt                => bool" ("_ |- _ [ok]"   [51,51   ]50)
```

translations

```
  "E⊢e :: T" == "(E,e,T) ∈ ty_expr"
  "E⊢e[::]T" == "(E,e,T) ∈ ty_exprs"
  "E⊢c √"    == "(E,c)   ∈ wt_stmt"
```

inductive *"ty_expr" "ty_exprs" "wt_stmt"* **intros**

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

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

constdefs

```
 wf_java_mdecl :: "java_mb prog => cname => java_mb mdecl => bool"
"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))"
```

syntax
```
 wf_java_prog :: "java_mb prog => bool"
```
translations
```
  "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"
```
apply `(simp add: wf_prog_def)`
apply `(simp add: wf_cdecl_def)`
apply `(erule conjE)+`
apply `(drule bspec, assumption)`
apply `simp`
apply `(erule conjE)+`
apply `(drule bspec, assumption)`
apply `(simp add: wf_mdecl_def split_beta)`
done

lemma `wt_is_type: "(E⊢e::T ⟶ wf_prog wf_mb (prg E) ⟶ is_type (prg E) T) ∧`
```
       (E⊢es[::]Ts ⟶ wf_prog wf_mb (prg E) ⟶ Ball (set Ts) (is_type (prg E))) ∧
       (E⊢c √ ⟶ True)"
```
apply `(rule ty_expr_ty_exprs_wt_stmt.induct)`
apply `auto`
apply `(   erule typeof_empty_is_type)`
apply `(  simp split add: split_if_asm)`
apply `( drule field_fields)`
apply `( drule (1) fields_is_type)`
apply `(  simp (no_asm_simp))`
apply `(assumption)`
apply `(auto dest!: max_spec2mheads method_wf_mdecl is_type_rTI`
```
            simp add: wf_mdecl_def)
```
done

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

end

theory `Exceptions = State:`

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

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

  start_heap :: "'c prog ⇒ aheap"
  "start_heap G ≡ empty (XcptRef NullPointer ↦ blank G (Xcpt NullPointer))
                        (XcptRef ClassCast ↦ blank G (Xcpt ClassCast))
                        (XcptRef OutOfMemory ↦ blank G (Xcpt OutOfMemory))"
```

consts

```
  cname_of :: "aheap ⇒ val ⇒ cname"

translations
  "cname_of hp v" == "fst (the (hp (the_Addr v)))"


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

lemma preallocatedD:
  "preallocated hp ⟹ ∃fs. hp (XcptRef x) = Some (Xcpt x, fs)"
  by (unfold preallocated_def) fast

lemma preallocatedE [elim?]:
  "preallocated hp ⟹ (⋀fs. hp (XcptRef x) = Some (Xcpt x, fs) ⟹ P hp) ⟹ P hp"
  by (fast dest: preallocatedD)

lemma cname_of_xcp:
  "raise_if b x None = Some xcp ⟹ preallocated hp
  ⟹ cname_of (hp::aheap) xcp = Xcpt x"
proof -
  assume "raise_if b x None = Some xcp"
  hence "xcp = Addr (XcptRef x)"
    by (simp add: raise_if_def split: split_if_asm)
  moreover
  assume "preallocated hp"
  then obtain fs where "hp (XcptRef x) = Some (Xcpt x, fs)" ..
  ultimately
  show ?thesis by simp
qed

lemma preallocated_start:
  "preallocated (start_heap G)"
  apply (unfold preallocated_def)
  apply (unfold start_heap_def)
  apply (rule allI)
  apply (case_tac x)
  apply (auto simp add: blank_def)
  done



end
```

2.11 Conformity Relations for Type Soundness Proof

theory *Conform = State + WellType + Exceptions:*

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

constdefs

```
  hext :: "aheap => aheap => bool" ("_ <=| _" [51,51] 50)
 "h<=|h' == ∀a C fs. h a = Some(C,fs) --> (∃fs'. h' a = Some(C,fs'))"

  conf :: "'c prog => aheap => val => ty => bool"
                                    ("_,_ |- _ ::<= _"  [51,51,51,51] 50)
 "G,h|-v::<=T == ∃T'. typeof (option_map obj_ty o h) v = Some T' ∧ G⊢T'≼T"

  lconf :: "'c prog => aheap => ('a ⇀ val) => ('a ⇀ ty) => bool"
                                    ("_,_ |- _ [::<=] _" [51,51,51,51] 50)
 "G,h|-vs[::<=]Ts == ∀n T. Ts n = Some T --> (∃v. vs n = Some v ∧ G,h|-v::<=T)"

  oconf :: "'c prog => aheap => obj => bool" ("_,_ |- _ [ok]" [51,51,51] 50)
 "G,h|-obj [ok] == G,h|-snd obj[::<=]map_of (fields (G,fst obj))"

  hconf :: "'c prog => aheap => bool" ("_ |-h _ [ok]" [51,51] 50)
 "G|-h h [ok]    == ∀a obj. h a = Some obj --> G,h|-obj [ok]"

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

  conforms :: "xstate => java_mb env_ => bool" ("_ ::<= _" [51,51] 50)
 "s::<=E == prg E|-h heap (store s) [ok] ∧
            prg E,heap (store s)|-locals (store s)[::<=]localT E ∧
            xconf (heap (store s)) (abrupt s)"
```

syntax *(xsymbols)*

```
  hext     :: "aheap => aheap => bool"
              ("_ ≤| _" [51,51] 50)

  conf     :: "'c prog => aheap => val => ty => bool"
              ("_,_ ⊢ _ ::≼ _" [51,51,51,51] 50)

  lconf    :: "'c prog => aheap => ('a ⇀ val) => ('a ⇀ ty) => bool"
              ("_,_ ⊢ _ [::≼] _" [51,51,51,51] 50)

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

  hconf    :: "'c prog => aheap => bool"
              ("_ ⊢h _ √" [51,51] 50)

  conforms :: "state => java_mb env_ => bool"
              ("_ ::≼ _" [51,51] 50)
```

2.11.1 hext

lemma *hextI:*
" ∀a C fs . h a = Some (C,fs) -->
(∃fs'. h' a = Some (C,fs')) ==> h≤|h'"
apply *(unfold hext_def)*
apply *auto*
done

lemma *hext_objD: "[|h≤|h'; h a = Some (C,fs) |] ==> ∃fs'. h' a = Some (C,fs')"*
apply *(unfold hext_def)*
apply *(force)*
done

lemma *hext_refl [simp]: "h≤|h"*
apply *(rule hextI)*
apply *(fast)*
done

lemma *hext_new [simp]: "h a = None ==> h≤|h(a↦x)"*
apply *(rule hextI)*
apply *auto*
done

lemma *hext_trans: "[|h≤|h'; h'≤|h''|] ==> h≤|h''"*
apply *(rule hextI)*
apply *(fast dest: hext_objD)*
done

lemma *hext_upd_obj: "h a = Some (C,fs) ==> h≤|h(a↦(C,fs'))"*
apply *(rule hextI)*
apply *auto*
done

2.11.2 conf

lemma *conf_Null [simp]: "G,h⊢Null::≤T = G⊢RefT NullT≤T"*
apply *(unfold conf_def)*
apply *(simp (no_asm))*
done

lemma *conf_litval [rule_format (no_asm), simp]:*
"typeof (λv. None) v = Some T --> G,h⊢v::≤T"
apply *(unfold conf_def)*
apply *(rule val.induct)*
apply *auto*
done

lemma *conf_AddrI: "[|h a = Some obj; G⊢obj_ty obj≤T|] ==> G,h⊢Addr a::≤T"*
apply *(unfold conf_def)*
apply *(simp)*
done

lemma *conf_obj_AddrI: "[|h a = Some (C,fs); G⊢C≤C D|] ==> G,h⊢Addr a::≤ Class D"*
apply *(unfold conf_def)*

```
apply (simp)
done

lemma defval_conf [rule_format (no_asm)]:
  "is_type G T --> G,h⊢default_val T::≼T"
apply (unfold conf_def)
apply (rule_tac "y" = "T" in ty.exhaust)
apply  (erule ssubst)
apply  (rule_tac "y" = "prim_ty" in prim_ty.exhaust)
apply    (auto simp add: widen.null)
done

lemma conf_upd_obj:
"h a = Some (C,fs) ==> (G,h(a↦(C,fs'))⊢x::≼T) = (G,h⊢x::≼T)"
apply (unfold conf_def)
apply (rule val.induct)
apply auto
done

lemma conf_widen [rule_format (no_asm)]:
  "wf_prog wf_mb G ==> G,h⊢x::≼T --> G⊢T≼T' --> G,h⊢x::≼T'"
apply (unfold conf_def)
apply (rule val.induct)
apply (auto intro: widen_trans)
done

lemma conf_hext [rule_format (no_asm)]: "h≤|h' ==> G,h⊢v::≼T --> G,h'⊢v::≼T"
apply (unfold conf_def)
apply (rule val.induct)
apply (auto dest: hext_objD)
done

lemma new_locD: "[|h a = None; G,h⊢Addr t::≼T|] ==> t≠a"
apply (unfold conf_def)
apply auto
done

lemma conf_RefTD [rule_format (no_asm)]:
 "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)"
apply (unfold conf_def)
apply(induct_tac "a'")
apply(auto)
done

lemma conf_NullTD: "G,h⊢a'::≼RefT NullT ==> a' = Null"
apply (drule conf_RefTD)
apply auto
done

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"
apply (drule conf_RefTD)
apply auto
```

done

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)"
apply *(fast dest: non_npD)*
done

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)"
apply*(rule_tac "y" = "t"* **in** *ref_ty.exhaust)*
apply *(fast dest: conf_NullTD)*
apply *(fast dest: non_np_objD)*
done

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'"
apply*(induct_tac "vs")*
apply*(clarsimp)*
apply*(clarsimp)*
apply*(frule list_all2_lengthD [THEN sym])*
apply*(simp (no_asm_use) add: length_Suc_conv)*
apply*(safe)*
apply*(frule list_all2_lengthD [THEN sym])*
apply*(simp (no_asm_use) add: length_Suc_conv)*
apply*(clarify)*
apply*(fast elim: conf_widen)*
done

2.11.3 lconf

lemma *lconfD: "[| G,h⊢vs[::≼]Ts; Ts n = Some T |] ==> G,h⊢(the (vs n))::≼T"*
apply *(unfold lconf_def)*
apply *(force)*
done

lemma *lconf_hext [elim]: "[| G,h⊢l[::≼]L; h≤|h' |] ==> G,h'⊢l[::≼]L"*
apply *(unfold lconf_def)*
apply *(fast elim: conf_hext)*
done

lemma *lconf_upd: "!!X. [| G,h⊢l[::≼]lT;*
G,h⊢v::≼T; lT va = Some T |] ==> G,h⊢l(va↦v)[::≼]lT"
apply *(unfold lconf_def)*
apply *auto*
done

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))"
apply*(induct_tac "fs")*
apply *auto*

done

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"
apply *(unfold lconf_def init_vars_def)*
apply *auto*
apply(*rule lconf_init_vars_lemma)*
apply(*erule_tac [3] asm_rl)*
apply(*intro strip)*
apply(*erule defval_conf)*
apply *auto*
done

lemma *lconf_ext: "[|G,s⊢l[::≼]L; G,s⊢v::≼T|] ==> G,s⊢l(vn↦v)[::≼]L(vn↦T)"*
apply *(unfold lconf_def)*
apply *auto*
done

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)"
apply *(unfold lconf_def)*
apply(*induct_tac "vns")*
apply(*clarsimp)*
apply(*clarsimp)*
apply(*frule list_all2_lengthD)*
apply(*auto simp add: length_Suc_conv)*
done

lemma *lconf_restr: "⟦lT vn = None; G, h ⊢ l [::≼] lT(vn↦T)⟧ ⟹ G, h ⊢ l [::≼] lT"*
apply *(unfold lconf_def)*
apply *(intro strip)*
apply *(case_tac "n = vn")*
apply *auto*
done

2.11.4 oconf

lemma *oconf_hext: "G,h⊢obj√ ==> h≤|h' ==> G,h'⊢obj√"*
apply *(unfold oconf_def)*
apply *(fast)*
done

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))"
apply *(unfold oconf_def lconf_def)*
apply *auto*
done

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

2.11.5 hconf

lemma *hconfD: "[|G⊢h h√; h a = Some obj|] ==> G,h⊢obj√"*
apply *(unfold hconf_def)*
apply *(fast)*
done

lemma *hconfI: "∀a obj. h a=Some obj --> G,h⊢obj√ ==> G⊢h h√"*
apply *(unfold hconf_def)*
apply *(fast)*
done

2.11.6 xconf

lemma *xconf_raise_if: "xconf h x ⟹ xconf h (raise_if b xcn x)"*
by *(simp add: xconf_def raise_if_def)*

2.11.7 conforms

lemma *conforms_heapD: "(x, (h, l))::≼(G, lT) ==> G⊢h h√"*
apply *(unfold conforms_def)*
apply *(simp)*
done

lemma *conforms_localD: "(x, (h, l))::≼(G, lT) ==> G,h⊢l[::≼]lT"*
apply *(unfold conforms_def)*
apply *(simp)*
done

lemma *conforms_xcptD: "(x, (h, l))::≼(G, lT) ==> xconf h x"*
apply *(unfold conforms_def)*
apply *(simp)*
done

lemma *conformsI: "[|G⊢h h√; G,h⊢l[::≼]lT; xconf h x|] ==> (x, (h, l))::≼(G, lT)"*
apply *(unfold conforms_def)*
apply *auto*
done

lemma *conforms_restr: "⟦lT vn = None; s ::≼ (G, lT(vn↦T)) ⟧ ⟹ s ::≼ (G, lT)"*
by *(simp add: conforms_def, fast intro: lconf_restr)*

lemma *conforms_xcpt_change: "⟦ (x, (h,l))::≼ (G, lT); xconf h x ⟶ xconf h x' ⟧ ⟹*
(x', (h,l))::≼ (G, lT)"
by *(simp add: conforms_def)*

lemma *preallocated_hext: "⟦ preallocated h; h≤|h'⟧ ⟹ preallocated h'"*
by *(simp add: preallocated_def hext_def)*

lemma *xconf_hext: "⟦ xconf h vo; h≤|h'⟧ ⟹ xconf h' vo"*
by *(simp add: xconf_def preallocated_def hext_def)*

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

by(*fast dest: conforms_localD conforms_xcptD elim!: conformsI xconf_hext)*

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)"
apply(*rule conforms_hext)*
apply *auto*
apply(*rule hconfI)*
apply(*drule conforms_heapD)*
apply(*tactic {* auto_tac (HOL_cs addEs [thm "oconf_hext"]*
*addDs [thm "hconfD"], simpset() delsimps [split_paired_All]) *})*
done

lemma *conforms_upd_local:*
"[|(x,(h, l))::≼(G, lT); G,h⊢v::≼T; lT va = Some T|]
  ==> (x,(h, l(va↦v)))::≼(G, lT)"
apply *(unfold conforms_def)*
apply(*auto elim: lconf_upd)*
done

end

Chapter 3

Java Virtual Machine

3.1 State of the JVM

theory *JVMState = Conform:*

3.1.1 Frame Stack

types
```
 opstack   = "val list"
 locvars   = "val list"
 p_count   = nat

 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

constdefs
```
  raise_system_xcpt :: "bool ⇒ xcpt ⇒ val option"
  "raise_system_xcpt b x ≡ raise_if b x None"
```

3.1.3 Runtime State

types
```
  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 -
  obtain *ref xp* **where** *"new_Addr hp = (ref, xp)"* **by** *(cases "new_Addr hp")*
  moreover
  assume *"snd (new_Addr hp) = Some xcp"*
  ultimately
  show *?thesis* **by** *(auto dest: new_AddrD)*
qed

end

3.2 Instructions of the JVM

theory *JVMInstructions = JVMState:*

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
```

types

```
  bytecode = "instr list"
  exception_entry = "p_count × p_count × p_count × cname"
                  — start-pc, end-pc, handler-pc, exception type
  exception_table = "exception_entry list"
  jvm_method = "nat × nat × bytecode × exception_table"
   — max stacksize, size of register set, instruction sequence, handler table
  jvm_prog = "jvm_method prog"
```

end

3.3 JVM Instruction Semantics

theory *JVMExecInstr = JVMInstructions + JVMState:*

consts

```
  exec_instr :: "[instr, jvm_prog, aheap, opstack, locvars,
                  cname, sig, p_count, frame list] => jvm_state"
```

primrec

```
 "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 arbitrary,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 = JVMInstructions:*

constdefs

```
  match_exception_entry :: "jvm_prog ⇒ cname ⇒ p_count ⇒ exception_entry ⇒ bool"
  "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"
```

consts

```
  match_exception_table :: "jvm_prog ⇒ cname ⇒ p_count ⇒ exception_table
                           ⇒ p_count option"
```

primrec

```
  "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)"
```

consts

```
  ex_table_of :: "jvm_method ⇒ exception_table"
```

translations

```
  "ex_table_of m" == "snd (snd (snd m))"
```

consts

```
  find_handler :: "jvm_prog ⇒ val option ⇒ aheap ⇒ frame list ⇒ jvm_state"
```

primrec

```
  "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))"
  by (induct et) (auto split: split_if_asm)
```

end

3.5 Program Execution in the JVM

theory *JVMExec = JVMExecInstr + JVMExceptions:*

consts
```
exec :: "jvm_prog × jvm_state => jvm_state option"
```

— exec is not recursive. recdef is just used for pattern matching
recdef *exec "{}"*
```
"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"
```

constdefs
```
exec_all :: "[jvm_prog,jvm_state,jvm_state] => bool"
            ("_ |- _ -jvm-> _" [61,61,61]60)
"G |- s -jvm-> t == (s,t) ∈ {(s,t). exec(G,s) = Some t}^*"
```

syntax *(xsymbols)*
```
exec_all :: "[jvm_prog,jvm_state,jvm_state] => bool"
            ("_ ⊢ _ -jvm→ _" [61,61,61]60)
```

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.

constdefs
```
start_state :: "jvm_prog ⇒ cname ⇒ mname ⇒ jvm_state"
"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 arbitrary, C, (m,[]), 0)])"
```

end

3.6 Example for generating executable code from JVM semantics

theory *JVMListExample = SystemClasses + JVMExec:*

consts

```
  list_nam :: cnam
  test_nam :: cnam
  append_name :: mname
  makelist_name :: mname
  val_nam :: vnam
  next_nam :: vnam
```

constdefs

```
  list_name :: cname
  "list_name == Cname list_nam"

  test_name :: cname
  "test_name == Cname test_nam"

  val_name :: vname
  "val_name == VName val_nam"

  next_name :: vname
  "next_name == VName next_nam"

  append_ins :: bytecode
  "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]"

  list_class :: "jvm_method class"
  "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)]))])"

  make_list_ins :: bytecode
  "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]"

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

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

types_code

```
  cnam ("string")
  vnam ("string")
  mname ("string")
  loc_ ("int")
```

consts_code

```
  "new_Addr" ("new'_addr {* %x. case x of None => True | Some y => False *}/ {* None *}/
{* Loc *}")

  "arbitrary" ("(raise ERROR)")
  "arbitrary" :: "val" ("{* Unit *}")
  "arbitrary" :: "cname" ("Object")

  "list_nam" (""list"")
  "test_nam" (""test"")
  "append_name" (""append"")
  "makelist_name" (""makelist"")
  "val_nam" (""val"")
  "next_nam" (""next"")
```

ML *{**

```
fun new_addr p none loc hp =
```

```
  let fun nr i = if p (hp (loc i)) then (loc i, none) else nr (i+1);
  in nr 0 end;
*}
```

3.6.1 Single step execution

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

ML {* test *}
ML {* exec (E, the it) *}
ML {* exec (E, the it) *}
ML {* exec (E, the it) *}
ML {* exec (E, the it) *}
ML {* exec (E, the it) *}
ML {* exec (E, the it) *}
ML {* exec (E, the it) *}
ML {* exec (E, the it) *}
ML {* exec (E, the it) *}
ML {* exec (E, the it) *}
ML {* exec (E, the it) *}
ML {* exec (E, the it) *}
ML {* exec (E, the it) *}
ML {* exec (E, the it) *}
ML {* exec (E, the it) *}
ML {* exec (E, the it) *}
ML {* exec (E, the it) *}
ML {* exec (E, the it) *}
ML {* exec (E, the it) *}
ML {* exec (E, the it) *}
ML {* exec (E, the it) *}
ML {* exec (E, the it) *}
ML {* exec (E, the it) *}
ML {* exec (E, the it) *}
ML {* exec (E, the it) *}
ML {* exec (E, the it) *}
ML {* exec (E, the it) *}
ML {* exec (E, the it) *}
ML {* exec (E, the it) *}
ML {* exec (E, the it) *}
ML {* exec (E, the it) *}
ML {* exec (E, the it) *}
ML {* exec (E, the it) *}
ML {* exec (E, the it) *}
ML {* exec (E, the it) *}
ML {* exec (E, the it) *}
ML {* exec (E, the it) *}
ML {* exec (E, the it) *}
ML {* exec (E, the it) *}
ML {* exec (E, the it) *}
ML {* exec (E, the it) *}
ML {* exec (E, the it) *}
ML {* exec (E, the it) *}
```

ML *{* exec (E, the it) *}*
ML *{* exec (E, the it) *}*
ML *{* exec (E, the it) *}*
ML *{* exec (E, the it) *}*
ML *{* exec (E, the it) *}*
ML *{* exec (E, the it) *}*
ML *{* exec (E, the it) *}*
ML *{* exec (E, the it) *}*
ML *{* exec (E, the it) *}*

end

3.7 A Defensive JVM

theory *JVMDefensive = JVMExec:*

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

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

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

consts *isAddr :: "val ⇒ bool"*
recdef *isAddr "{}"*
"isAddr (Addr loc) = True"
"isAddr v = False"

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

constdefs
isRef :: "val ⇒ bool"
"isRef v ≡ v = Null ∨ isAddr v"

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

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

"check_instr (LitPush v) G hp stk vars Cl sig pc frs =
(¬isAddr v)"

"check_instr (New C) G hp stk vars Cl sig pc frs =
is_class G C"

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

  "check_instr (Invoke C mn ps) G hp stk vars Cl sig pc 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 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 frs =
  (0 < length stk)"

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

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

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

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

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

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

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

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

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

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


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


consts
  "exec_all_d" :: "jvm_prog ⇒ jvm_state type_error ⇒ jvm_state type_error ⇒ bool"
                   ("_ |- _ -jvmd-> _" [61,61,61]60)

syntax (xsymbols)
  "exec_all_d" :: "jvm_prog ⇒ jvm_state type_error ⇒ jvm_state type_error ⇒ bool"
                   ("_ ⊢ _ -jvmd→ _" [61,61,61]60)

defs
  exec_all_d_def:
  "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"
  by (cases P, auto)

lemma exec_d_no_errorI [intro]:
  "check G s ⟹ exec_d G (Normal s) ≠ TypeError"
  by (unfold exec_d_def) simp

theorem no_type_error_commutes:
  "exec_d G (Normal s) ≠ TypeError ⟹
  exec_d G (Normal s) = Normal (exec (G, s))"
  by (unfold exec_d_def, auto)


lemma defensive_imp_aggressive:
  "G ⊢ (Normal s) -jvmd→ (Normal t) ⟹ G ⊢ s -jvm→ t"
proof -
  have "⋀x y. G ⊢ x -jvmd→ y ⟹ ∀s t. x = Normal s ⟶ y = Normal t ⟶  G ⊢ s -jvm→
t"
    apply (unfold exec_all_d_def)
    apply (erule rtrancl_induct)
     apply (simp add: exec_all_def)
    apply (fold exec_all_d_def)
    apply simp
    apply (intro allI impI)
```

```
    apply (erule disjE, simp)
    apply (elim exE conjE)
    apply (erule allE, erule impE, assumption)
    apply (simp add: exec_all_def exec_d_def split: type_error.splits split_if_asm)
    apply (rule rtrancl_trans, assumption)
    apply blast
    done
  moreover
  assume "G ⊢ (Normal s) -jvmd→ (Normal t)"
  ultimately
  show "G ⊢ s -jvm→ t" by blast
qed

end
```

Chapter 4

Bytecode Verifier

4.1 Semilattices

theory *Semilat = While_Combinator:*

types *'a ord = "'a ⇒ 'a ⇒ bool"*
'a binop = "'a ⇒ 'a ⇒ 'a"
*'a sl = "'a set * 'a ord * 'a binop"*

consts
"@lesub" :: "'a ⇒ 'a ord ⇒ 'a ⇒ bool" ("(_ /<='__ _)" [50, 1000, 51] 50)
"@lesssub" :: "'a ⇒ 'a ord ⇒ 'a ⇒ bool" ("(_ /<'__ _)" [50, 1000, 51] 50)
defs
lesub_def: "x <=_r y == r x y"
lesssub_def: "x <_r y == x <=_r y & x ~= y"

syntax *(xsymbols)*
"@lesub" :: "'a ⇒ 'a ord ⇒ 'a ⇒ bool" ("(_ /≤_ _)" [50, 1000, 51] 50)

consts
"@plussub" :: "'a ⇒ ('a ⇒ 'b ⇒ 'c) ⇒ 'b ⇒ 'c" ("(_ /+'__ _)" [65, 1000, 66] 65)
defs
plussub_def: "x +_f y == f x y"

syntax *(xsymbols)*
"@plussub" :: "'a ⇒ ('a ⇒ 'b ⇒ 'c) ⇒ 'b ⇒ 'c" ("(_ /+_ _)" [65, 1000, 66] 65)

syntax *(xsymbols)*
"@plussub" :: "'a ⇒ ('a ⇒ 'b ⇒ 'c) ⇒ 'b ⇒ 'c" ("(_ /⊔_ _)" [65, 1000, 66] 65)

constdefs
ord :: "('a'a)set ⇒ 'a ord"*
"ord r == %x y. (x,y):r"

order :: "'a ord ⇒ bool"
"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)"

acc :: "'a ord ⇒ bool"
"acc r == wf{(y,x) . x <_r y}"

top :: "'a ord ⇒ 'a ⇒ bool"
"top r T == !x. x <=_r T"

closed :: "'a set ⇒ 'a binop ⇒ bool"
"closed A f == !x:A. !y:A. x +_f y : A"

semilat :: "'a sl ⇒ bool"
"semilat == %(A,r,f). order r & closed A f &
(!x:A. !y:A. x <=_r x +_f y) &
(!x:A. !y:A. y <=_r x +_f y) &
(!x:A. !y:A. !z:A. x <=_r z & y <=_r z ⟶ x +_f y <=_r z)"

```
 is_ub :: "('a*'a)set ⇒ 'a ⇒ 'a ⇒ 'a ⇒ bool"
"is_ub r x y u == (x,u):r & (y,u):r"

 is_lub :: "('a*'a)set ⇒ 'a ⇒ 'a ⇒ 'a ⇒ bool"
"is_lub r x y u == is_ub r x y u & (!z. is_ub r x y z ⟶ (u,z):r)"

 some_lub :: "('a*'a)set ⇒ 'a ⇒ 'a ⇒ 'a"
"some_lub r x y == SOME z. is_lub r x y z"

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

lemma order_refl [simp, intro]:
  "order r ⟹ x <=_r x"
  by (simp add: order_def)

lemma order_antisym:
  "⟦ order r; x <=_r y; y <=_r x ⟧ ⟹ x = y"
apply (unfold order_def)
apply (simp (no_asm_simp))
done

lemma order_trans:
   "⟦ order r; x <=_r y; y <=_r z ⟧ ⟹ x <=_r z"
apply (unfold order_def)
apply blast
done

lemma order_less_irrefl [intro, simp]:
   "order r ⟹ ~ x <_r x"
apply (unfold order_def lesssub_def)
apply blast
done

lemma order_less_trans:
  "⟦ order r; x <_r y; y <_r z ⟧ ⟹ x <_r z"
apply (unfold order_def lesssub_def)
apply blast
done

lemma topD [simp, intro]:
  "top r T ⟹ x <=_r T"
  by (simp add: top_def)

lemma top_le_conv [simp]:
  "⟦ order r; top r T ⟧ ⟹ (T <=_r x) = (x = T)"
  by (blast intro: order_antisym)

lemma semilat_Def:
"semilat(A,r,f) == order r & closed A f &
                (!x:A. !y:A. x <=_r x +_f y) &
```

```
                (!x:A. !y:A. y <=_r x +_f y) &
                (!x:A. !y:A. !z:A. x <=_r z & y <=_r z ⟶ x +_f y <=_r z)"
apply (unfold semilat_def split_conv [THEN eq_reflection])
apply (rule refl [THEN eq_reflection])
done

lemma (in semilat) orderI [simp, intro]:
  "order r"
  by (insert semilat) (simp add: semilat_Def)

lemma (in semilat) closedI [simp, intro]:
  "closed A f"
  by (insert semilat) (simp add: semilat_Def)

lemma closedD:
  "⟦ closed A f; x:A; y:A ⟧ ⟹ x +_f y : A"
  by (unfold closed_def) blast

lemma closed_UNIV [simp]: "closed UNIV f"
  by (simp add: closed_def)


lemma (in semilat) closed_f [simp, intro]:
  "⟦x:A; y:A⟧ ⟹ x +_f y : A"
  by (simp add: closedD [OF closedI])

lemma (in semilat) refl_r [intro, simp]:
  "x <=_r x"
  by simp

lemma (in semilat) antisym_r [intro?]:
  "⟦ x <=_r y; y <=_r x ⟧ ⟹ x = y"
  by (rule order_antisym) auto

lemma (in semilat) trans_r [trans, intro?]:
  "⟦x <=_r y; y <=_r z⟧ ⟹ x <=_r z"
  by (auto intro: order_trans)


lemma (in semilat) ub1 [simp, intro?]:
  "⟦ x:A; y:A ⟧ ⟹ x <=_r x +_f y"
  by (insert semilat) (unfold semilat_Def, simp)

lemma (in semilat) ub2 [simp, intro?]:
  "⟦ x:A; y:A ⟧ ⟹ y <=_r x +_f y"
  by (insert semilat) (unfold semilat_Def, simp)

lemma (in semilat) lub [simp, intro?]:
 "⟦ x <=_r z; y <=_r z; x:A; y:A; z:A ⟧ ⟹ x +_f y <=_r z"
  by (insert semilat) (unfold semilat_Def, simp)


lemma (in semilat) plus_le_conv [simp]:
  "⟦ x:A; y:A; z:A ⟧ ⟹ (x +_f y <=_r z) = (x <=_r z & y <=_r z)"
```

```
  by (blast intro: ub1 ub2 lub order_trans)

lemma (in semilat) le_iff_plus_unchanged:
  "⟦ x:A; y:A ⟧ ⟹ (x <=_r y) = (x +_f y = y)"
apply (rule iffI)
 apply (blast intro: antisym_r refl_r lub ub2)
apply (erule subst)
apply simp
done

lemma (in semilat) le_iff_plus_unchanged2:
  "⟦ x:A; y:A ⟧ ⟹ (x <=_r y) = (y +_f x = y)"
apply (rule iffI)
 apply (blast intro: order_antisym lub order_refl ub1)
apply (erule subst)
apply simp
done


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 -
  from a b have ab: "a +_f b ∈ A" ..
  from this c have abc: "(a +_f b) +_f c ∈ A" ..
  from b c have bc: "b +_f c ∈ A" ..
  from a this have abc': "a +_f (b +_f c) ∈ A" ..

  show ?thesis
  proof
    show "a +_f (b +_f c) <=_r (a +_f b) +_f c"
    proof -
      from a b have "a <=_r a +_f b" ..
      also from ab c have "... <=_r ... +_f c" ..
      finally have "a<": "a <=_r (a +_f b) +_f c" .
      from a b have "b <=_r a +_f b" ..
      also from ab c have "... <=_r ... +_f c" ..
      finally have "b<": "b <=_r (a +_f b) +_f c" .
      from ab c have "c<": "c <=_r (a +_f b) +_f c" ..
      from "b<" "c<" b c abc have "b +_f c <=_r (a +_f b) +_f c" ..
      from "a<" this a bc abc show ?thesis ..
    qed
    show "(a +_f b) +_f c <=_r a +_f (b +_f c)"
    proof -
      from b c have "b <=_r b +_f c" ..
      also from a bc have "... <=_r a +_f ..." ..
      finally have "b<": "b <=_r a +_f (b +_f c)" .
      from b c have "c <=_r b +_f c" ..
      also from a bc have "... <=_r a +_f ..." ..
      finally have "c<": "c <=_r a +_f (b +_f c)" .
      from a bc have "a<": "a <=_r a +_f (b +_f c)" ..
      from "a<" "b<" a b abc' have "a +_f b <=_r a +_f (b +_f c)" ..
      from this "c<" ab c abc' show ?thesis ..
    qed
```

```
  qed
qed

lemma (in semilat) plus_com_lemma:
  "⟦a ∈ A; b ∈ A⟧ ⟹ a +_f b <=_r b +_f a"
proof -
  assume a: "a ∈ A" and b: "b ∈ A"
  from b a have "a <=_r b +_f a" ..
  moreover from b a have "b <=_r b +_f a" ..
  moreover note a b
  moreover from b a have "b +_f a ∈ A" ..
  ultimately show ?thesis ..
qed

lemma (in semilat) plus_commutative:
  "⟦a ∈ A; b ∈ A⟧ ⟹ a +_f b = b +_f a"
by(blast intro: order_antisym plus_com_lemma)

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)"
  by (simp add: is_lub_def)

lemma is_ubI:
  "⟦ (x,u) : r; (y,u) : r ⟧ ⟹ is_ub r x y u"
  by (simp add: is_ub_def)

lemma is_ubD:
  "is_ub r x y u ⟹ (x,u) : r & (y,u) : r"
  by (simp add: is_ub_def)


lemma is_lub_bigger1 [iff]:
  "is_lub (r^* ) x y y = ((x,y):r^* )"
apply (unfold is_lub_def is_ub_def)
apply blast
done

lemma is_lub_bigger2 [iff]:
  "is_lub (r^* ) x y x = ((y,x):r^* )"
apply (unfold is_lub_def is_ub_def)
apply blast
done

lemma extend_lub:
  "⟦ single_valued r; is_lub (r^* ) x y u; (x',x) : r ⟧
  ⟹ EX v. is_lub (r^* ) x' y v"
apply (unfold is_lub_def is_ub_def)
apply (case_tac "(y,x) : r^*")
 apply (case_tac "(y,x') : r^*")
  apply blast
 apply (blast elim: converse_rtranclE dest: single_valuedD)
apply (rule exI)
apply (rule conjI)
 apply (blast intro: converse_rtrancl_into_rtrancl dest: single_valuedD)
```

```
apply (blast intro: rtrancl_into_rtrancl converse_rtrancl_into_rtrancl
             elim: converse_rtranclE dest: single_valuedD)
done

lemma single_valued_has_lubs [rule_format]:
  "⟦ single_valued r; (x,u) : r^* ⟧ ⟹ (!y. (y,u) : r^* ⟶
  (EX z. is_lub (r^* ) x y z))"
apply (erule converse_rtrancl_induct)
 apply clarify
 apply (erule converse_rtrancl_induct)
  apply blast
 apply (blast intro: converse_rtrancl_into_rtrancl)
apply (blast intro: extend_lub)
done

lemma some_lub_conv:
  "⟦ acyclic r; is_lub (r^* ) x y u ⟧ ⟹ some_lub (r^* ) x y = u"
apply (unfold some_lub_def is_lub_def)
apply (rule someI2)
 apply assumption
apply (blast intro: antisymD dest!: acyclic_impl_antisym_rtrancl)
done

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)"
  by (fastsimp dest: single_valued_has_lubs simp add: some_lub_conv)
```

4.1.1 An executable lub-finder

```
constdefs
 exec_lub :: "('a * 'a) set ⇒ ('a ⇒ 'a) ⇒ 'a binop"
"exec_lub r f x y == while (λz. (x,z) ∉ r^*) f y"


lemma acyclic_single_valued_finite:
 "⟦acyclic r; single_valued r; (x,y) ∈ r^*⟧
  ⟹ finite (r ∩ {a. (x, a) ∈ r^*} × {b. (b, y) ∈ r^*})"
apply(erule converse_rtrancl_induct)
 apply(rule_tac B = "{}" in finite_subset)
  apply(simp only:acyclic_def)
  apply(blast intro:rtrancl_into_trancl2 rtrancl_trancl_trancl)
 apply simp
apply(rename_tac x x')
apply(subgoal_tac "r ∩ {a. (x,a) ∈ r^*} × {b. (b,y) ∈ r^*} =
                   insert (x,x') (r ∩ {a. (x', a) ∈ r^*} × {b. (b, y) ∈ r^*})")
 apply simp
apply(blast intro:converse_rtrancl_into_rtrancl
            elim:converse_rtranclE dest:single_valuedD)
done


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"
apply(unfold exec_lub_def)
apply(rule_tac P = "λz. (y,z) ∈ r* ∧ (z,u) ∈ r*" and
               r = "(r ∩ {(a,b). (y,a) ∈ r* ∧ (b,u) ∈ r*})^-1" in while_rule)
    apply(blast dest: is_lubD is_ubD)
   apply(erule conjE)
   apply(erule_tac z = u in converse_rtranclE)
    apply(blast dest: is_lubD is_ubD)
   apply(blast dest:rtrancl_into_rtrancl)
  apply(rename_tac s)
  apply(subgoal_tac "is_ub (r*) x y s")
   prefer 2 apply(simp add:is_ub_def)
  apply(subgoal_tac "(u, s) ∈ r*")
   prefer 2 apply(blast dest:is_lubD)
  apply(erule converse_rtranclE)
   apply blast
  apply(simp only:acyclic_def)
  apply(blast intro:rtrancl_into_trancl2 rtrancl_trancl_trancl)
 apply(rule finite_acyclic_wf)
  apply simp
  apply(erule acyclic_single_valued_finite)
   apply(blast intro:single_valuedI)
  apply(simp add:is_lub_def is_ub_def)
 apply simp
 apply(erule acyclic_subset)
 apply blast
apply simp
apply(erule conjE)
apply(erule_tac z = u in converse_rtranclE)
 apply(blast dest: is_lubD is_ubD)
apply(blast dest:rtrancl_into_rtrancl)
done

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)"
  by (fastsimp dest: single_valued_has_lubs simp add: exec_lub_conv)

end
```

4.2 The Error Type

theory *Err = Semilat:*

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

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

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

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

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

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

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

err :: "'a set ⇒ 'a err set"
"err A == insert Err {x . ? y:A. x = OK y}"

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

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

syntax
err_semilat :: "'a esl ⇒ bool"
translations
"err_semilat L" == "semilat(Err.sl L)"

consts
strict :: "('a ⇒ 'b err) ⇒ ('a err ⇒ 'b err)"
primrec
"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)"

```
  by (cases x, auto)

lemma not_Err_eq:
  "(x ≠ Err) = (∃a. x = OK a)"
  by (cases x) auto

lemma not_OK_eq:
  "(∀y. x ≠ OK y) = (x = Err)"
  by (cases x) auto

lemma unfold_lesub_err:
  "e1 <=_(le r) e2 == le r e1 e2"
  by (simp add: lesub_def)

lemma le_err_refl:
  "!x. x <=_r x ⟹ e <=_(Err.le r) e"
apply (unfold lesub_def Err.le_def)
apply (simp split: err.split)
done

lemma le_err_trans [rule_format]:
  "order r ⟹ e1 <=_(le r) e2 ⟶ e2 <=_(le r) e3 ⟶ e1 <=_(le r) e3"
apply (unfold unfold_lesub_err le_def)
apply (simp split: err.split)
apply (blast intro: order_trans)
done

lemma le_err_antisym [rule_format]:
  "order r ⟹ e1 <=_(le r) e2 ⟶ e2 <=_(le r) e1 ⟶ e1=e2"
apply (unfold unfold_lesub_err le_def)
apply (simp split: err.split)
apply (blast intro: order_antisym)
done

lemma OK_le_err_OK:
  "(OK x <=_(le r) OK y) = (x <=_r y)"
  by (simp add: unfold_lesub_err le_def)

lemma order_le_err [iff]:
  "order(le r) = order r"
apply (rule iffI)
 apply (subst order_def)
 apply (blast dest: order_antisym OK_le_err_OK [THEN iffD2]
              intro: order_trans OK_le_err_OK [THEN iffD1])
apply (subst order_def)
apply (blast intro: le_err_refl le_err_trans le_err_antisym
             dest: order_refl)
done

lemma le_Err [iff]:  "e <=_(le r) Err"
  by (simp add: unfold_lesub_err le_def)

lemma Err_le_conv [iff]:
 "Err <=_(le r) e  = (e = Err)"
```

by *(simp add: unfold_lesub_err le_def split: err.split)*

lemma *le_OK_conv [iff]:*
"e <=_(le r) OK x = (? y. e = OK y & y <=_r x)"
by *(simp add: unfold_lesub_err le_def split: err.split)*

lemma *OK_le_conv:*
"OK x <=_(le r) e = (e = Err | (? y. e = OK y & x <=_r y))"
by *(simp add: unfold_lesub_err le_def split: err.split)*

lemma *top_Err [iff]: "top (le r) Err"*
by *(simp add: top_def)*

lemma *OK_less_conv [rule_format, iff]:*
"OK x <_(le r) e = (e=Err | (? y. e = OK y & x <_r y))"
by *(simp add: lesssub_def lesub_def le_def split: err.split)*

lemma *not_Err_less [rule_format, iff]:*
"~(Err <_(le r) x)"
by *(simp add: lesssub_def lesub_def le_def split: err.split)*

lemma *semilat_errI [intro]:* **includes** *semilat*
shows *"semilat(err A, Err.le r, lift2(%x y. OK(f x y)))"*
apply*(insert semilat)*
apply *(unfold semilat_Def closed_def plussub_def lesub_def*
lift2_def Err.le_def err_def)
apply *(simp split: err.split)*
done

lemma *err_semilat_eslI_aux:*
includes *semilat* **shows** *"err_semilat(esl(A,r,f))"*
apply *(unfold sl_def esl_def)*
apply *(simp add: semilat_errI[OF semilat])*
done

lemma *err_semilat_eslI [intro, simp]:*
"⋀L. semilat L ⟹ err_semilat(esl L)"
by*(simp add: err_semilat_eslI_aux split_tupled_all)*

lemma *acc_err [simp, intro!]: "acc r ⟹ acc(le r)"*
apply *(unfold acc_def lesub_def le_def lesssub_def)*
apply *(simp add: wf_eq_minimal split: err.split)*
apply *clarify*
apply *(case_tac "Err : Q")*
apply *blast*
apply *(erule_tac x = "{a . OK a : Q}"* **in** *allE)*
apply *(case_tac "x")*
apply *fast*
apply *blast*
done

lemma *Err_in_err [iff]: "Err : err A"*
by *(simp add: err_def)*

lemma *Ok_in_err [iff]: "(OK x : err A) = (x:A)"*
by *(auto simp add: err_def)*

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"
apply *(unfold lift_def)*
apply *(simp split: err.split)*
apply *blast*
done

lemma *Err_lift2 [simp]:*
"Err +_(lift2 f) x = Err"
by *(simp add: lift2_def plussub_def)*

lemma *lift2_Err [simp]:*
"x +_(lift2 f) Err = Err"
by *(simp add: lift2_def plussub_def split: err.split)*

lemma *OK_lift2_OK [simp]:*
"OK x +_(lift2 f) OK y = x +_f y"
by *(simp add: lift2_def plussub_def split: err.split)*

4.2.2 sup

lemma *Err_sup_Err [simp]:*
"Err +_(Err.sup f) x = Err"
by *(simp add: plussub_def Err.sup_def Err.lift2_def)*

lemma *Err_sup_Err2 [simp]:*
"x +_(Err.sup f) Err = Err"
by *(simp add: plussub_def Err.sup_def Err.lift2_def split: err.split)*

lemma *Err_sup_OK [simp]:*
"OK x +_(Err.sup f) OK y = OK(x +_f y)"
by *(simp add: plussub_def Err.sup_def Err.lift2_def)*

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)"
apply *(unfold Err.sup_def lift2_def plussub_def)*
apply *(rule iffI)*
apply *(simp split: err.split_asm)*
apply *clarify*
apply *simp*
done

lemma *Err_sup_eq_Err [iff]:*
"(Err.sup f ex ey = Err) = (ex=Err | ey=Err)"
apply *(unfold Err.sup_def lift2_def plussub_def)*
apply *(simp split: err.split)*
done

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"
  by *(blast intro: semilat.le_iff_plus_unchanged [THEN iffD1]*
  semilat.le_iff_plus_unchanged2 [THEN iffD1])

lemma *semilat_le_err_plus_Err [simp]:*
  "⟦ x: err A; semilat(err A, le r, f) ⟧ ⟹ x +_f Err = Err"
  by *(blast intro: semilat.le_iff_plus_unchanged [THEN iffD1]*
  semilat.le_iff_plus_unchanged2 [THEN iffD1])

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"
apply *(rule OK_le_err_OK [THEN iffD1])*
apply *(erule subst)*
apply *(simp add:semilat.ub1)*
done

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"
apply *(rule OK_le_err_OK [THEN iffD1])*
apply *(erule subst)*
apply *(simp add:semilat.ub2)*
done

lemma *eq_order_le:*
  "⟦ x=y; order r ⟧ ⟹ x <=_r y"
apply *(unfold order_def)*
apply *blast*
done

lemma *OK_plus_OK_eq_Err_conv [simp]:*
  "⟦ x:A; y:A; semilat(err A, le r, fe) ⟧ ⟹
  ((OK x) +_fe (OK y) = Err) = (~(? z:A. x <=_r z & y <=_r z))"
proof -
  have *plus_le_conv3: "⋀A x y z f r.*
    ⟦ semilat (A,r,f); x +_f y <=_r z; x:A; y:A; z:A ⟧
    ⟹ x <=_r z ∧ y <=_r z"
    by *(rule semilat.plus_le_conv [THEN iffD1])*
  case *rule_context*
  thus *?thesis*
  apply *(rule_tac iffI)*
   apply *clarify*
   apply *(drule OK_le_err_OK [THEN iffD2])*
   apply *(drule OK_le_err_OK [THEN iffD2])*
   apply *(drule semilat.lub[of _ _ _ "OK x" _ "OK y"])*
        apply *assumption*
       apply *assumption*
      apply *simp*
     apply *simp*
    apply *simp*

```
   apply simp
  apply (case_tac "(OK x) +_fe (OK y)")
   apply assumption
  apply (rename_tac z)
  apply (subgoal_tac "OK z: err A")
  apply (drule eq_order_le)
    apply (erule semilat.orderI)
   apply (blast dest: plus_le_conv3)
  apply (erule subst)
  apply (blast intro: semilat.closedI closedD)
  done
qed
```

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))"
  by blast

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(Union AS)) (lift2 f)"
apply (unfold closed_def err_def)
apply simp
apply clarify
apply simp
apply fast
done
```

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(Union AS, r, f)"
apply (unfold semilat_def sl_def)
apply (simp add: closed_err_Union_lift2I)
apply (rule conjI)
 apply blast
apply (simp add: err_def)
apply (rule conjI)
 apply clarify
 apply (rename_tac A a u B b)
 apply (case_tac "A = B")
  apply simp
 apply simp
apply (rule conjI)
 apply clarify
 apply (rename_tac A a u B b)
 apply (case_tac "A = B")
  apply simp
```

```
 apply simp
apply clarify
apply (rename_tac A ya yb B yd z C c a b)
apply (case_tac "A = B")
 apply (case_tac "A = C")
  apply simp
 apply (rotate_tac -1)
 apply simp
apply (rotate_tac -1)
apply (case_tac "B = C")
 apply simp
apply (rotate_tac -1)
apply simp
done

end
```

4.3 Fixed Length Lists

theory *Listn = Err:*

constdefs

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

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

syntax *"@lesublist" :: "'a list ⇒ 'a ord ⇒ 'a list ⇒ bool"*
("(_ /<=[_] _)" [50, 0, 51] 50)
syntax *"@lesssublist" :: "'a list ⇒ 'a ord ⇒ 'a list ⇒ bool"*
("(_ /<[_] _)" [50, 0, 51] 50)
translations
"x <=[r] y" == "x <=_(Listn.le r) y"
"x <[r] y" == "x <_(Listn.le r) y"

constdefs
map2 :: "('a ⇒ 'b ⇒ 'c) ⇒ 'a list ⇒ 'b list ⇒ 'c list"
"map2 f == (%xs ys. map (split f) (zip xs ys))"

syntax *"@plussublist" :: "'a list ⇒ ('a ⇒ 'b ⇒ 'c) ⇒ 'b list ⇒ 'c list"*
("(_ /+[_] _)" [65, 0, 66] 65)
translations *"x +[f] y" == "x +_(map2 f) y"*

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

constdefs
sl :: "nat ⇒ 'a sl ⇒ 'a list sl"
"sl n == %(A,r,f). (list n A, le r, map2 f)"

sup :: "('a ⇒ 'b ⇒ 'c err) ⇒ 'a list ⇒ 'b list ⇒ 'c list err"
"sup f == %xs ys. if size xs = size ys then coalesce(xs +[f] ys) else Err"

upto_esl :: "nat ⇒ 'a esl ⇒ 'a list esl"
"upto_esl m == %(A,r,f). (Union{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"
by *(simp add: lesub_def)*

lemma *Nil_le_conv [iff]:*
"([] <=[r] ys) = (ys = [])"
apply *(unfold lesub_def Listn.le_def)*
apply *simp*
done

lemma *Cons_notle_Nil [iff]:*
"~ x#xs <=[r] []"
apply *(unfold lesub_def Listn.le_def)*
apply *simp*
done

lemma *Cons_le_Cons [iff]:*
"x#xs <=[r] y#ys = (x <=_r y & xs <=[r] ys)"
apply *(unfold lesub_def Listn.le_def)*
apply *simp*
done

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)"
apply *(unfold lesssub_def)*
apply *blast*
done

lemma *list_update_le_cong:*
"⟦ i<size xs; xs <=[r] ys; x <=_r y ⟧ ⟹ xs[i:=x] <=[r] ys[i:=y]"
apply *(unfold unfold_lesub_list)*
apply *(unfold Listn.le_def)*
apply *(simp add: list_all2_conv_all_nth nth_list_update)*
done

lemma *le_listD:*
"⟦ xs <=[r] ys; p < size xs ⟧ ⟹ xs!p <=_r ys!p"
apply *(unfold Listn.le_def lesub_def)*
apply *(simp add: list_all2_conv_all_nth)*
done

lemma *le_list_refl:*
"!x. x <=_r x ⟹ xs <=[r] xs"
apply *(unfold unfold_lesub_list)*
apply *(simp add: Listn.le_def list_all2_conv_all_nth)*
done

lemma *le_list_trans:*
"⟦ order r; xs <=[r] ys; ys <=[r] zs ⟧ ⟹ xs <=[r] zs"
apply *(unfold unfold_lesub_list)*
apply *(simp add: Listn.le_def list_all2_conv_all_nth)*
apply *clarify*
apply *simp*
apply *(blast intro: order_trans)*
done

lemma *le_list_antisym:*
"⟦ order r; xs <=[r] ys; ys <=[r] xs ⟧ ⟹ xs = ys"
apply *(unfold unfold_lesub_list)*

```
apply (simp add: Listn.le_def list_all2_conv_all_nth)
apply (rule nth_equalityI)
 apply blast
apply clarify
apply simp
apply (blast intro: order_antisym)
done

lemma order_listI [simp, intro!]:
  "order r ⟹ order(Listn.le r)"
apply (subst order_def)
apply (blast intro: le_list_refl le_list_trans le_list_antisym
             dest: order_refl)
done


lemma lesub_list_impl_same_size [simp]:
  "xs <=[r] ys ⟹ size ys = size xs"
apply (unfold Listn.le_def lesub_def)
apply (simp add: list_all2_conv_all_nth)
done

lemma lesssub_list_impl_same_size:
  "xs <_(Listn.le r) ys ⟹ size ys = size xs"
apply (unfold lesssub_def)
apply auto
done

lemma le_list_appendI:
  "⋀b c d. a <=[r] b ⟹ c <=[r] d ⟹ a@c <=[r] b@d"
apply (induct a)
 apply simp
apply (case_tac b)
apply auto
done

lemma le_listI:
  "length a = length b ⟹ (⋀n. n < length a ⟹ a!n <=_r b!n) ⟹ a <=[r] b"
  apply (unfold lesub_def Listn.le_def)
  apply (simp add: list_all2_conv_all_nth)
  done

lemma listI:
  "⟦ length xs = n; set xs <= A ⟧ ⟹ xs : list n A"
apply (unfold list_def)
apply blast
done

lemma listE_length [simp]:
   "xs : list n A ⟹ length xs = n"
apply (unfold list_def)
apply blast
done
```

lemma *less_lengthI:*
"⟦ xs : list n A; p < n ⟧ ⟹ p < length xs"
by *simp*

lemma *listE_set [simp]:*
"xs : list n A ⟹ set xs <= A"
apply *(unfold list_def)*
apply *blast*
done

lemma *list_0 [simp]:*
"list 0 A = {[]}"
apply *(unfold list_def)*
apply *auto*
done

lemma *in_list_Suc_iff:*
"(xs : list (Suc n) A) = (? y:A. ? ys:list n A. xs = y#ys)"
apply *(unfold list_def)*
apply *(case_tac "xs")*
apply *auto*
done

lemma *Cons_in_list_Suc [iff]:*
"(x#xs : list (Suc n) A) = (x:A & xs : list n A)"
apply *(simp add: in_list_Suc_iff)*
done

lemma *list_not_empty:*
"? a. a:A ⟹ ? xs. xs : list n A"
apply *(induct "n")*
apply *simp*
apply *(simp add: in_list_Suc_iff)*
apply *blast*
done

lemma *nth_in [rule_format, simp]:*
"!i n. length xs = n ⟶ set xs <= A ⟶ i < n ⟶ (xs!i) : A"
apply *(induct "xs")*
apply *simp*
apply *(simp add: nth_Cons split: nat.split)*
done

lemma *listE_nth_in:*
"⟦ xs : list n A; i < n ⟧ ⟹ (xs!i) : A"
by *auto*

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

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 -
  have "⋀n. a@b ∈ list n A ⟹ ∃n1 n2. n=n1+n2 ∧ a ∈ list n1 A ∧ b ∈ list n2 A"
    (is "⋀n. ?list a n ⟹ ∃n1 n2. ?P a n n1 n2")
  proof (induct a)
    fix n assume "?list [] n"
    hence "?P [] n 0 n" by simp
    thus "∃n1 n2. ?P [] n n1 n2" by fast
  next
    fix n l ls
    assume "?list (l#ls) n"
    then obtain n' where n: "n = Suc n'" "l ∈ A" and "ls@b ∈ list n' A" by fastsimp
    assume "⋀n. ls @ b ∈ list n A ⟹ ∃n1 n2. n = n1 + n2 ∧ ls ∈ list n1 A ∧ b ∈ list
n2 A"
    hence "∃n1 n2. n' = n1 + n2 ∧ ls ∈ list n1 A ∧ b ∈ list n2 A" .
    then obtain n1 n2 where "n' = n1 + n2" "ls ∈ list n1 A" "b ∈ list n2 A" by fast
    with n have "?P (l#ls) n (n1+1) n2" by simp
    thus "∃n1 n2. ?P (l#ls) n n1 n2" by fastsimp
  qed
  moreover
  assume "a@b ∈ list n A" "⋀n1 n2. n=n1+n2 ⟹ a ∈ list n1 A ⟹ b ∈ list n2 A ⟹
P"
  ultimately
  show ?thesis by blast
qed


lemma listt_update_in_list [simp, intro!]:
  "⟦ xs : list n A; x:A ⟧ ⟹ xs[i := x] : list n A"
apply (unfold list_def)
apply simp
done

lemma plus_list_Nil [simp]:
  "[] +[f] xs = []"
apply (unfold plussub_def map2_def)
apply simp
done

lemma plus_list_Cons [simp]:
  "(x#xs) +[f] ys = (case ys of [] ⇒ [] | y#ys ⇒ (x +_f y)#(xs +[f] ys))"
  by (simp add: plussub_def map2_def split: list.split)

lemma length_plus_list [rule_format, simp]:
  "!ys. length(xs +[f] ys) = min(length xs) (length ys)"
apply (induct xs)
 apply simp
apply clarify
apply (simp (no_asm_simp) split: list.split)
done

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)"
apply (induct n)
 apply simp
apply clarify
apply (case_tac xs)
 apply simp
apply (force simp add: nth_Cons split: list.split nat.split)
done


lemma (in semilat) plus_list_ub1 [rule_format]:
 "⟦ set xs <= A; set ys <= A; size xs = size ys ⟧
  ⟹ xs <=[r] xs +[f] ys"
apply (unfold unfold_lesub_list)
apply (simp add: Listn.le_def list_all2_conv_all_nth)
done

lemma (in semilat) plus_list_ub2:
 "⟦set xs <= A; set ys <= A; size xs = size ys ⟧
  ⟹ ys <=[r] xs +[f] ys"
apply (unfold unfold_lesub_list)
apply (simp add: Listn.le_def list_all2_conv_all_nth)
done

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"
apply (unfold unfold_lesub_list)
apply (simp add: Listn.le_def list_all2_conv_all_nth)
done

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])"
apply (unfold unfold_lesub_list)
apply (simp add: Listn.le_def list_all2_conv_all_nth)
apply (induct xs)
 apply simp
apply (simp add: in_list_Suc_iff)
apply clarify
apply (simp add: nth_Cons split: nat.split)
done

lemma acc_le_listI [intro!]:
  "⟦ order r; acc r ⟧ ⟹ acc(Listn.le r)"
apply (unfold acc_def)
apply (subgoal_tac
 "wf(UN n. {(ys,xs). size xs = n & size ys = n & xs <_(Listn.le r) ys})")
 apply (erule wf_subset)
 apply (blast intro: lesssub_list_impl_same_size)
apply (rule wf_UN)
 prefer 2
 apply clarify
```

```
 apply (rename_tac m n)
 apply (case_tac "m=n")
  apply simp
 apply (rule conjI)
  apply (fast intro!: equals0I dest: not_sym)
 apply (fast intro!: equals0I dest: not_sym)
apply clarify
apply (rename_tac n)
apply (induct_tac n)
 apply (simp add: lesssub_def cong: conj_cong)
apply (rename_tac k)
apply (simp add: wf_eq_minimal)
apply (simp (no_asm) add: length_Suc_conv cong: conj_cong)
apply clarify
apply (rename_tac M m)
apply (case_tac "? x xs. size xs = k & x#xs : M")
 prefer 2
 apply (erule thin_rl)
 apply (erule thin_rl)
 apply blast
apply (erule_tac x = "{a. ? xs. size xs = k & a#xs:M}" in allE)
apply (erule impE)
 apply blast
apply (thin_tac "? x xs. ?P x xs")
apply clarify
apply (rename_tac maxA xs)
apply (erule_tac x = "{ys. size ys = size xs & maxA#ys : M}" in allE)
apply (erule impE)
 apply blast
apply clarify
apply (thin_tac "m : M")
apply (thin_tac "maxA#xs : M")
apply (rule bexI)
 prefer 2
 apply assumption
apply clarify
apply simp
apply blast
done

lemma closed_listI:
  "closed S f ⟹ closed (list n S) (map2 f)"
apply (unfold closed_def)
apply (induct n)
 apply simp
apply clarify
apply (simp add: in_list_Suc_iff)
apply clarify
apply simp
done


lemma Listn_sl_aux:
includes semilat shows "semilat (Listn.sl n (A,r,f))"
```

apply *(unfold Listn.sl_def)*
apply *(simp (no_asm) only: semilat_Def split_conv)*
apply *(rule conjI)*
 apply *simp*
apply *(rule conjI)*
 apply *(simp only: closedI closed_listI)*
apply *(simp (no_asm) only: list_def)*
apply *(simp (no_asm_simp) add: plus_list_ub1 plus_list_ub2 plus_list_lub)*
done

lemma *Listn_sl: "⋀L. semilat L ⟹ semilat (Listn.sl n L)"*
 by*(simp add: Listn_sl_aux split_tupled_all)*

lemma *coalesce_in_err_list [rule_format]:*
  "!xes. xes : list n (err A) ⟶ coalesce xes : err(list n A)"
apply *(induct n)*
 apply *simp*
apply *clarify*
apply *(simp add: in_list_Suc_iff)*
apply *clarify*
apply *(simp (no_asm) add: plussub_def Err.sup_def lift2_def split: err.split)*
apply *force*
done

lemma *lem: "⋀x xs. x +_(op #) xs = x#xs"*
  by *(simp add: plussub_def)*

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))"
apply *(induct n)*
  apply *simp*
apply *clarify*
apply *(simp add: in_list_Suc_iff)*
apply *clarify*
apply *(simp split: err.split_asm add: lem Err.sup_def lift2_def)*
apply *(force simp add: semilat_le_err_OK1)*
done

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))"
apply *(induct n)*
 apply *simp*
apply *clarify*
apply *(simp add: in_list_Suc_iff)*
apply *clarify*
apply *(simp split: err.split_asm add: lem Err.sup_def lift2_def)*
apply *(force simp add: semilat_le_err_OK2)*
done

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"
apply (unfold semilat_Def plussub_def err_def)
apply (simp add: lift2_def)
apply clarify
apply (rotate_tac -3)
apply (erule thin_rl)
apply (erule thin_rl)
apply force
done

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))"
apply (induct n)
 apply simp
apply clarify
apply (simp add: in_list_Suc_iff)
apply clarify
apply (simp (no_asm_use) split: err.split_asm add: lem Err.sup_def lift2_def)
apply clarify
apply (rule conjI)
 apply (blast intro: lift2_le_ub)
apply blast
done

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)"
  by (simp add: OK_plus_OK_eq_Err_conv [THEN iffD1])


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))"
apply (induct n)
 apply simp
apply clarify
apply (simp add: in_list_Suc_iff)
apply clarify
apply (simp split: err.split_asm add: lem Err.sup_def lift2_def)
 apply (blast dest: lift2_eq_ErrD)
done

lemma closed_err_lift2_conv:
  "closed (err A) (lift2 f) = (!x:A. !y:A. x +_f y : err A)"
apply (unfold closed_def)
apply (simp add: err_def)
done
```

```
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))"
apply (unfold map2_def)
apply (induct n)
 apply simp
apply clarify
apply (simp add: in_list_Suc_iff)
apply clarify
apply (simp add: plussub_def closed_err_lift2_conv)
done

lemma closed_lift2_sup:
  "closed (err A) (lift2 f) ⟹
  closed (err (list n A)) (lift2 (sup f))"
  by (fastsimp  simp add: closed_def plussub_def sup_def lift2_def
                          coalesce_in_err_list closed_map2_list
               split: err.split)

lemma err_semilat_sup:
  "err_semilat (A,r,f) ⟹
  err_semilat (list n A, Listn.le r, sup f)"
apply (unfold Err.sl_def)
apply (simp only: split_conv)
apply (simp (no_asm) only: semilat_Def plussub_def)
apply (simp (no_asm_simp) only: semilat.closedI closed_lift2_sup)
apply (rule conjI)
 apply (drule semilat.orderI)
 apply simp
apply (simp (no_asm) only: unfold_lesub_err Err.le_def err_def sup_def lift2_def)
apply (simp (no_asm_simp) add: coalesce_eq_OK1_D coalesce_eq_OK2_D split: err.split)
apply (blast intro: coalesce_eq_OK_ub_D dest: coalesce_eq_Err_D)
done

lemma err_semilat_upto_esl:
  "⋀L. err_semilat L ⟹ err_semilat(upto_esl m L)"
apply (unfold Listn.upto_esl_def)
apply (simp (no_asm_simp) only: split_tupled_all)
apply simp
apply (fastsimp intro!: err_semilat_UnionI err_semilat_sup
               dest: lesub_list_impl_same_size
               simp add: plussub_def Listn.sup_def)
done

end
```

4.4 Typing and Dataflow Analysis Framework

theory *Typing_Framework = Listn:*

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

types

```
  's step_type = "nat ⇒ 's ⇒ (nat × 's) list"
```

constdefs

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

 stables :: "'s ord ⇒ 's step_type ⇒ 's list ⇒ bool"
"stables r step ss == !p<size ss. stable r step ss p"

 is_bcv :: "'s ord ⇒ 's ⇒ 's step_type
           ⇒ nat ⇒ 's set ⇒ ('s list ⇒ 's list) ⇒ bool"
"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)"

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

end

4.5 Products as Semilattices

theory *Product = Err:*

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

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

 esl :: "'a esl ⇒ 'b esl ⇒ ('a * 'b ) esl"
"esl == %(A,rA,fA) (B,rB,fB). (A <*> B, le rA rB, sup fA fB)"
```

syntax *"@lesubprod" :: "'a*'b ⇒ 'a ord ⇒ 'b ord ⇒ 'b ⇒ bool"*
("(_ /<='(_,_') _)" [50, 0, 0, 51] 50)
translations *"p <=(rA,rB) q" == "p <=_(Product.le rA rB) q"*

lemma *unfold_lesub_prod:*
"p <=(rA,rB) q == le rA rB p q"
by *(simp add: lesub_def)*

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

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)"
apply *(unfold lesssub_def)*
apply *simp*
apply *blast*
done

lemma *order_le_prod [iff]:*
"order(Product.le rA rB) = (order rA & order rB)"
apply *(unfold order_def)*
apply *simp*
apply *blast*
done

lemma *acc_le_prodI [intro!]:*
"⟦ acc rA; acc rB ⟧ ⟹ acc(Product.le rA rB)"
apply *(unfold acc_def)*
apply *(rule wf_subset)*
apply *(erule wf_lex_prod)*
apply *assumption*
apply *(auto simp add: lesssub_def less_prod_Pair_conv lex_prod_def)*
done

lemma *closed_lift2_sup:*
"⟦ closed (err A) (lift2 f); closed (err B) (lift2 g) ⟧ ⟹

```
  closed (err(A<*>B)) (lift2(sup f g))"
apply (unfold closed_def plussub_def lift2_def err_def sup_def)
apply (simp split: err.split)
apply blast
done

lemma unfold_plussub_lift2:
  "e1 +_(lift2 f) e2 == lift2 f e1 e2"
  by (simp add: plussub_def)


lemma plus_eq_Err_conv [simp]:
  "⟦ x:A; y:A; semilat(err A, Err.le r, lift2 f) ⟧
  ⟹ (x +_f y = Err) = (~(? z:A. x <=_r z & y <=_r z))"
proof -
  have plus_le_conv2:
    "⋀r f z. ⟦ z : err A; semilat (err A, r, f); OK x : err A; OK y : err A;
                OK x +_f OK y <=_r z⟧ ⟹ OK x <=_r z ∧ OK y <=_r z"
    by (rule semilat.plus_le_conv [THEN iffD1])
  case rule_context
  thus ?thesis
  apply (rule_tac iffI)
   apply clarify
   apply (drule OK_le_err_OK [THEN iffD2])
   apply (drule OK_le_err_OK [THEN iffD2])
   apply (drule semilat.lub[of _ _ _ "OK x" _ "OK y"])
        apply assumption
       apply assumption
      apply simp
     apply simp
    apply simp
   apply simp
  apply (case_tac "x +_f y")
   apply assumption
  apply (rename_tac "z")
  apply (subgoal_tac "OK z: err A")
  apply (frule plus_le_conv2)
       apply assumption
      apply simp
      apply blast
     apply simp
    apply (blast dest: semilat.orderI order_refl)
   apply blast
  apply (erule subst)
  apply (unfold semilat_def err_def closed_def)
  apply simp
  done
qed

lemma err_semilat_Product_esl:
  "⋀L1 L2. ⟦ err_semilat L1; err_semilat L2 ⟧ ⟹ err_semilat(Product.esl L1 L2)"
apply (unfold esl_def Err.sl_def)
apply (simp (no_asm_simp) only: split_tupled_all)
apply simp
```

```
apply (simp (no_asm) only: semilat_Def)
apply (simp (no_asm_simp) only: semilat.closedI closed_lift2_sup)
apply (simp (no_asm) only: unfold_lesub_err Err.le_def unfold_plussub_lift2 sup_def)
apply (auto elim: semilat_le_err_OK1 semilat_le_err_OK2
            simp add: lift2_def  split: err.split)
apply (blast dest: semilat.orderI)
apply (blast dest: semilat.orderI)

apply (rule OK_le_err_OK [THEN iffD1])
apply (erule subst, subst OK_lift2_OK [symmetric], rule semilat.lub)
apply simp
apply simp
apply simp
apply simp
apply simp
apply simp

apply (rule OK_le_err_OK [THEN iffD1])
apply (erule subst, subst OK_lift2_OK [symmetric], rule semilat.lub)
apply simp
apply simp
apply simp
apply simp
apply simp
apply simp
done

end
```

4.6 More on Semilattices

theory *SemilatAlg = Typing_Framework + Product:*

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

consts
```
 "@plusplussub" :: "'a list ⇒ ('a ⇒ 'a ⇒ 'a) ⇒ 'a ⇒ 'a" ("(_ /++'__ _)" [65, 1000,
66] 65)
```
primrec
```
  "[] ++_f y = y"
  "(x#xs) ++_f y = xs ++_f (x +_f y)"
```

constdefs
```
 bounded :: "'s step_type ⇒ nat ⇒ bool"
"bounded step n == !p<n. !s. !(q,t):set(step p s). q<n"

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

 mono :: "'s ord ⇒ 's step_type ⇒ nat ⇒ 's set ⇒ bool"
"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"
```
by *(unfold pres_type_def, blast)*

lemma *monoD:*
```
  "⟦ mono r step n A; p < n; s∈A; s <=_r t ⟧ ⟹ step p s <=|r| step p t"
```
by *(unfold mono_def, blast)*

lemma *boundedD:*
```
  "⟦ bounded step n; p < n; (q,t) : set (step p xs) ⟧ ⟹ q < n"
```
by *(unfold bounded_def, blast)*

lemma *lesubstep_type_refl [simp, intro]:*
```
  "(⋀x. x <=_r x) ⟹ x <=|r| x"
```
by *(unfold lesubstep_type_def) auto*

lemma *lesub_step_typeD:*
```
  "a <=|r| b ⟹ (x,y) ∈ set a ⟹ ∃y'. (x, y') ∈ set b ∧ y <=_r y'"
```
by *(unfold lesubstep_type_def) blast*

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"
```

apply *(unfold Listn.le_def lesub_def semilat_def)*
apply *(simp add: list_all2_conv_all_nth nth_list_update)*
done

lemma *plusplus_closed:* **includes** *semilat* **shows**
"⋀y. ⟦ set x ⊆ A; y ∈ A⟧ ⟹ x ++_f y ∈ A"
proof *(induct x)*
show *"⋀y. y ∈ A ⟹ [] ++_f y ∈ A"* **by** *simp*
fix *y x xs*
assume *y: "y ∈ A"* **and** *xs: "set (x#xs) ⊆ A"*
assume *IH: "⋀y. ⟦ set xs ⊆ A; y ∈ A⟧ ⟹ xs ++_f y ∈ A"*
from *xs* **obtain** *x: "x ∈ A"* **and** *"set xs ⊆ A"* **by** *simp*
from *x y* **have** *"(x +_f y) ∈ A"* **..**
with *xs* **have** *"xs ++_f (x +_f y) ∈ A"* **by** *- (rule IH)*
thus *"(x#xs) ++_f y ∈ A"* **by** *simp*
qed

lemma *(in semilat) pp_ub2:*
"⋀y. ⟦ set x ⊆ A; y ∈ A⟧ ⟹ y <=_r x ++_f y"
proof *(induct x)*
from *semilat* **show** *"⋀y. y <=_r [] ++_f y"* **by** *simp*

fix *y a l*
assume *y: "y ∈ A"*
assume *"set (a#l) ⊆ A"*
then obtain *a: "a ∈ A"* **and** *x: "set l ⊆ A"* **by** *simp*
assume *"⋀y. ⟦set l ⊆ A; y ∈ A⟧ ⟹ y <=_r l ++_f y"*
hence *IH: "⋀y. y ∈ A ⟹ y <=_r l ++_f y"* **.**

from *a y* **have** *"y <=_r a +_f y"* **..**
also from *a y* **have** *"a +_f y ∈ A"* **..**
hence *"(a +_f y) <=_r l ++_f (a +_f y)"* **by** *(rule IH)*
finally have *"y <=_r l ++_f (a +_f y)"* **.**
thus *"y <=_r (a#l) ++_f y"* **by** *simp*
qed

lemma *(in semilat) pp_ub1:*
shows *"⋀y. ⟦set ls ⊆ A; y ∈ A; x ∈ set ls⟧ ⟹ x <=_r ls ++_f y"*
proof *(induct ls)*
show *"⋀y. x ∈ set [] ⟹ x <=_r [] ++_f y"* **by** *simp*

fix *y s ls*
assume *"set (s#ls) ⊆ A"*
then obtain *s: "s ∈ A"* **and** *ls: "set ls ⊆ A"* **by** *simp*
assume *y: "y ∈ A"*

assume
"⋀y. ⟦set ls ⊆ A; y ∈ A; x ∈ set ls⟧ ⟹ x <=_r ls ++_f y"
hence *IH: "⋀y. x ∈ set ls ⟹ y ∈ A ⟹ x <=_r ls ++_f y"* **.**

assume *"x ∈ set (s#ls)"*
then obtain *xls: "x = s ∨ x ∈ set ls"* **by** *simp*

```
  moreover {
    assume xs: "x = s"
    from s y have "s <=_r s +_f y" ..
    also from s y have "s +_f y ∈ A" ..
    with ls have "(s +_f y) <=_r ls ++_f (s +_f y)" by (rule pp_ub2)
    finally have "s <=_r ls ++_f (s +_f y)" .
    with xs have "x <=_r ls ++_f (s +_f y)" by simp
  }
  moreover {
    assume "x ∈ set ls"
    hence "⋀y. y ∈ A ⟹ x <=_r ls ++_f y" by (rule IH)
    moreover from s y have "s +_f y ∈ A" ..
    ultimately have "x <=_r ls ++_f (s +_f y)" .
  }
  ultimately
  have "x <=_r ls ++_f (s +_f y)" by blast
  thus "x <=_r (s#ls) ++_f y" by simp
qed


lemma (in semilat) pp_lub:
  assumes "z ∈ A"
  shows
  "⋀y. y ∈ A ⟹ set xs ⊆ A ⟹ ∀x ∈ set xs. x <=_r z ⟹ y <=_r z ⟹ xs ++_f y <=_r
z"
proof (induct xs)
  fix y assume "y <=_r z" thus "[] ++_f y <=_r z" by simp
next
  fix y l ls assume y: "y ∈ A" and "set (l#ls) ⊆ A"
  then obtain l: "l ∈ A" and ls: "set ls ⊆ A" by auto
  assume "∀x ∈ set (l#ls). x <=_r z"
  then obtain "l <=_r z" and lsz: "∀x ∈ set ls. x <=_r z" by auto
  assume "y <=_r z" have "l +_f y <=_r z" ..
  moreover
  from l y have "l +_f y ∈ A" ..
  moreover
  assume "⋀y. y ∈ A ⟹ set ls ⊆ A ⟹ ∀x ∈ set ls. x <=_r z ⟹ y <=_r z
          ⟹ ls ++_f y <=_r z"
  ultimately
  have "ls ++_f (l +_f y) <=_r z" using ls lsz by -
  thus "(l#ls) ++_f y <=_r z" by simp
qed


lemma ub1': includes semilat
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 -
  let "b <=_r ?map ++_f y" = ?thesis

  assume "y ∈ A"
  moreover
  assume "∀(p,s) ∈ set S. s ∈ A"
  hence "set ?map ⊆ A" by auto
```

```
  moreover
  assume "(a,b) ∈ set S"
  hence "b ∈ set ?map" by (induct S, auto)
  ultimately
  show ?thesis by - (rule pp_ub1)
qed



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"
apply (induct S)
apply auto
done


end
```

4.7 Kildall's Algorithm

theory *Kildall = SemilatAlg + While_Combinator:*

consts
```
 iter :: "'s binop ⇒ 's step_type ⇒
          's list ⇒ nat set ⇒ 's list × nat set"
 propa :: "'s binop ⇒ (nat × 's) list ⇒ 's list ⇒ nat set ⇒ 's list * nat set"
```

primrec
```
"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')"
```

defs *iter_def:*
```
"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)"
```

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

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

consts *merges :: "'s binop ⇒ (nat × 's) list ⇒ 's list ⇒ 's list"*
primrec
```
"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 [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 *(induct ps)*
  show *"⋀ss. ?P ss []"* **by** *simp*

  fix *ss p' ps'*
  assume *ss: "ss ∈ list n A"*
  assume *l: "p < length ss"*
  assume *"?steptype (p'#ps')"*
  then obtain *a b* **where**
    p': "p'=(a,b)" **and** *ab: "a<n" "b∈A"* **and** *"?steptype ps'"*
    by *(cases p', auto)*

```
  assume "⋀ss. p< length ss ⟹ ss ∈ list n A ⟹ ?steptype ps' ⟹ ?P ss ps'"
  hence IH: "⋀ss. ss ∈ list n A ⟹ p < length ss ⟹ ?P ss ps'" .

  from ss ab
  have "ss[a := b +_f ss!a] ∈ list n A" by (simp add: closedD)
  moreover
  from calculation
  have "p < length (ss[a := b +_f ss!a])" by simp
  ultimately
  have "?P (ss[a := b +_f ss!a]) ps'" by (rule IH)
  with p' l
  show "?P ss (p'#ps')" by simp
qed



lemma length_merges [rule_format, simp]:
  "∀ss. size(merges f ps ss) = size ss"
  by (induct_tac ps, auto)


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"
apply (insert closedI)
apply (unfold closed_def)
apply (induct_tac ps)
 apply simp
apply clarsimp
done

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"
by (simp add: merges_preserves_type_lemma)

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"
apply (induct_tac ps)
 apply simp
apply simp
apply clarify
apply (rule order_trans)
  apply simp
 apply (erule list_update_incr)
  apply simp
 apply simp
apply (blast intro!: listE_set intro: closedD listE_length [THEN nth_in])
done

lemma (in semilat) merges_incr:
 "⟦ xs ∈ list n A; ∀(p,x)∈set ps. p<size xs ∧ x ∈ A ⟧
```

```
  ⟹ xs <=[r] merges f ps xs"
  by (simp add: merges_incr_lemma)


lemma (in semilat) merges_same_conv [rule_format]:
 "(∀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))"
  apply (induct_tac ps)
   apply simp
  apply clarsimp
  apply (rename_tac p x ps xs)
  apply (rule iffI)
   apply (rule context_conjI)
    apply (subgoal_tac "xs[p := x +_f xs!p] <=[r] xs")
     apply (force dest!: le_listD simp add: nth_list_update)
    apply (erule subst, rule merges_incr)
      apply (blast intro!: listE_set intro: closedD listE_length [THEN nth_in])
     apply clarify
     apply (rule conjI)
      apply simp
      apply (blast dest: boundedD)
     apply blast
   apply clarify
   apply (erule allE)
   apply (erule impE)
    apply assumption
   apply (drule bspec)
    apply assumption
   apply (simp add: le_iff_plus_unchanged [THEN iffD1] list_update_same_conv [THEN iffD2])
   apply blast
  apply clarify
  apply (simp add: le_iff_plus_unchanged [THEN iffD1] list_update_same_conv [THEN iffD2])
  done


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"
  apply(insert semilat)
  apply (unfold Listn.le_def lesub_def semilat_def)
  apply (simp add: list_all2_conv_all_nth nth_list_update)
  done

lemma (in semilat) merges_pres_le_ub:
shows "⟦ set ts <= A; set ss <= A;
        ∀(p,t)∈set ps. t <=_r ts!p ∧ t ∈ A ∧ p < size ts; ss <=[r] ts ⟧
  ⟹ merges f ps ss <=[r] ts"
proof -
  { fix t ts ps
    have
    "⋀qs. ⟦set ts <= A; ∀(p,t)∈set ps. t <=_r ts!p ∧ t ∈ A ∧ p< size ts ⟧ ⟹
    set qs <= set ps  ⟶
    (∀ss. set ss <= A ⟶ ss <=[r] ts ⟶ merges f qs ss <=[r] ts)"
    apply (induct_tac qs)
```

```
     apply simp
    apply (simp (no_asm_simp))
    apply clarify
    apply (rotate_tac -2)
    apply simp
    apply (erule allE, erule impE, erule_tac [2] mp)
     apply (drule bspec, assumption)
     apply (simp add: closedD)
    apply (drule bspec, assumption)
    apply (simp add: list_update_le_listI)
    done
  } note this [dest]

  case rule_context
  thus ?thesis by blast
qed
```

```
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)"
  apply (induct qs)
   apply simp
  apply (simp (no_asm))
  apply clarify
  apply simp
  apply (rule conjI)
   apply (simp add: nth_list_update)
   apply blast
  apply (simp add: nth_list_update)
  apply blast
  done
```

```
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"
  apply (unfold stable_def)
  apply (subgoal_tac "∀s'. (q,s') ∈ set (step p (ss!p)) ⟶ s' : A")
   prefer 2
   apply clarify
   apply (erule pres_typeD)
    prefer 3 apply assumption
    apply (rule listE_nth_in)
     apply assumption
```

```
    apply simp
   apply simp
  apply simp
  apply clarify
  apply (subst nth_merges)
       apply simp
       apply (blast dest: boundedD)
      apply assumption
     apply clarify
     apply (rule conjI)
      apply (blast dest: boundedD)
     apply (erule pres_typeD)
       prefer 3 apply assumption
      apply simp
     apply simp
apply(subgoal_tac "q < length ss")
prefer 2 apply simp
  apply (frule nth_merges [of q _ _ "step p (ss!p)"])
apply assumption
  apply clarify
  apply (rule conjI)
   apply (blast dest: boundedD)
  apply (erule pres_typeD)
     prefer 3 apply assumption
    apply simp
   apply simp
  apply (drule_tac P = "λx. (a, b) ∈ set (step q x)" in subst)
   apply assumption

 apply (simp add: plusplus_empty)
 apply (cases "q ∈ w")
  apply simp
  apply (rule ub1')
     apply assumption
    apply clarify
    apply (rule pres_typeD)
       apply assumption
      prefer 3 apply assumption
     apply (blast intro: listE_nth_in dest: boundedD)
    apply (blast intro: pres_typeD dest: boundedD)
   apply (blast intro: listE_nth_in dest: boundedD)
  apply assumption

 apply simp
 apply (erule allE, erule impE, assumption, erule impE, assumption)
 apply (rule order_trans)
   apply simp
  defer
 apply (rule pp_ub2)
   apply simp
   apply clarify
   apply simp
   apply (rule pres_typeD)
      apply assumption
```

```
    prefer 3 apply assumption
   apply (blast intro: listE_nth_in dest: boundedD)
  apply (blast intro: pres_typeD dest: boundedD)
 apply (blast intro: listE_nth_in dest: boundedD)
apply blast
done


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"
 apply (unfold stable_def)
 apply (rule merges_pres_le_ub)
    apply simp
   apply simp
  prefer 2 apply assumption

 apply clarsimp
 apply (drule boundedD, assumption+)
 apply (erule allE, erule impE, assumption)
 apply (drule bspec, assumption)
 apply simp

 apply (drule monoD [of _ _ _ _ p "ss!p"  "ts!p"])
    apply assumption
   apply simp
  apply (simp add: le_listD)

 apply (drule lesub_step_typeD, assumption)
 apply clarify
 apply (drule bspec, assumption)
 apply simp
 apply (blast intro: order_trans)
 done

lemma termination_lemma: includes semilat
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"
apply(insert semilat)
 apply (unfold lesssub_def)
 apply (simp (no_asm_simp) add: merges_incr)
 apply (rule impI)
 apply (rule merges_same_conv [THEN iffD1, elim_format])
 apply assumption+
   defer
   apply (rule sym, assumption)
  defer apply simp
  apply (subgoal_tac "∀q t. ¬((q, t) ∈ set qs ∧ t +_f ss ! q ≠ ss ! q)")
  apply (blast intro!: psubsetI elim: equalityE)
  apply clarsimp
  apply (drule bspec, assumption)
```

```
   apply (drule bspec, assumption)
   apply clarsimp
  done

lemma iter_properties[rule_format]: includes semilat
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)"
apply(insert semilat)
apply (unfold iter_def stables_def)
apply (rule_tac P = "%(ss,w).
 ss ∈ list n A ∧ (∀p<n. p ∉ w ⟶ stable r step ss p) ∧ ss0 <=[r] ss ∧
 (∀ts∈list n A. ss0 <=[r] ts ∧ stables r step ts ⟶ ss <=[r] ts) ∧
 (∀p∈w. p < n)" and
 r = "{(ss',ss) . ss <[r] ss'} <*lex*> finite_psubset"
       in while_rule)

— Invariant holds initially:
apply (simp add:stables_def)

— Invariant is preserved:
apply(simp add: stables_def split_paired_all)
apply(rename_tac ss w)
apply(subgoal_tac "(SOME p. p ∈ w) ∈ w")
 prefer 2 apply (fast intro: someI)
apply(subgoal_tac "∀(q,t) ∈ set (step (SOME p. p ∈ w) (ss ! (SOME p. p ∈ w))). q < length
ss ∧ t ∈ A")
 prefer 2
 apply clarify
 apply (rule conjI)
  apply(clarsimp, blast dest!: boundedD)
 apply (erule pres_typeD)
  prefer 3
  apply assumption
  apply (erule listE_nth_in)
  apply blast
 apply blast
apply (subst decomp_propa)
 apply blast
apply simp
apply (rule conjI)
 apply (rule merges_preserves_type)
 apply blast
 apply clarify
 apply (rule conjI)
  apply(clarsimp, blast dest!: boundedD)
 apply (erule pres_typeD)
  prefer 3
  apply assumption
  apply (erule listE_nth_in)
```

```
  apply blast
 apply blast
apply (rule conjI)
 apply clarify
 apply (blast intro!: stable_pres_lemma)
apply (rule conjI)
 apply (blast intro!: merges_incr intro: le_list_trans)
apply (rule conjI)
 apply clarsimp
 apply (blast intro!: merges_bounded_lemma)
apply (blast dest!: boundedD)
```

— Postcondition holds upon termination:

```
apply(clarsimp simp add: stables_def split_paired_all)
```

— Well-foundedness of the termination relation:

```
apply (rule wf_lex_prod)
 apply (insert orderI [THEN acc_le_listI])
 apply (simp only: acc_def lesssub_def)
apply (rule wf_finite_psubset)
```

— Loop decreases along termination relation:

```
apply(simp add: stables_def split_paired_all)
apply(rename_tac ss w)
apply(subgoal_tac "(SOME p. p ∈ w) ∈ w")
 prefer 2 apply (fast intro: someI)
apply(subgoal_tac "∀(q,t) ∈ set (step (SOME p. p ∈ w) (ss ! (SOME p. p ∈ w))). q < length
ss ∧ t ∈ A")
 prefer 2
 apply clarify
 apply (rule conjI)
  apply(clarsimp, blast dest!: boundedD)
 apply (erule pres_typeD)
  prefer 3
  apply assumption
  apply (erule listE_nth_in)
  apply blast
 apply blast
apply (subst decomp_propa)
 apply blast
apply clarify
apply (simp del: listE_length
    add: lex_prod_def finite_psubset_def
         bounded_nat_set_is_finite)
apply (rule termination_lemma)
apply assumption+
defer
apply assumption
apply clarsimp
done
```

lemma *kildall_properties:* **includes** *semilat*

```
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)"
apply (unfold kildall_def)
apply(case_tac "iter f step ss0 (unstables r step ss0)")
apply(simp)
apply (rule iter_properties)
by (simp_all add: unstables_def stable_def)


lemma is_bcv_kildall: includes semilat
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)"
apply(unfold is_bcv_def wt_step_def)
apply(insert semilat kildall_properties[of A])
apply(simp add:stables_def)
apply clarify
apply(subgoal_tac "kildall r f step ss ∈ list n A")
 prefer 2 apply (simp(no_asm_simp))
apply (rule iffI)
 apply (rule_tac x = "kildall r f step ss" in bexI)
  apply (rule conjI)
   apply (blast)
  apply (simp  (no_asm_simp))
 apply(assumption)
apply clarify
apply(subgoal_tac "kildall r f step ss!p <=_r ts!p")
 apply simp
apply (blast intro!: le_listD less_lengthI)
done

end
```

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

theory *Typing_Framework_err = Typing_Framework + SemilatAlg:*

constdefs

```
wt_err_step :: "'s ord ⇒ 's err step_type ⇒ 's err list ⇒ bool"
"wt_err_step r step ts ≡ wt_step (Err.le r) Err step ts"

wt_app_eff :: "'s ord ⇒ (nat ⇒ 's ⇒ bool) ⇒ 's step_type ⇒ 's list ⇒ bool"
"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)"

map_snd :: "('b ⇒ 'c) ⇒ ('a × 'b) list ⇒ ('a × 'c) list"
"map_snd f ≡ map (λ(x,y). (x, f y))"

error :: "nat ⇒ (nat × 'a err) list"
"error n ≡ map (λx. (x,Err)) [0..n(]"

err_step :: "nat ⇒ (nat ⇒ 's ⇒ bool) ⇒ 's step_type ⇒ 's err step_type"
"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"

app_mono :: "'s ord ⇒ (nat ⇒ 's ⇒ bool) ⇒ nat ⇒ 's set ⇒ bool"
"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"
```
apply *(simp add: bounded_def err_step_def)*
apply *(erule allE, erule impE, assumption)*
apply *(erule_tac x = "OK a" in allE, drule bspec)*
apply *(simp add: map_snd_def)*
apply *fast*
apply *simp*
done

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

```
  apply (auto simp add: map_snd_def)
  done


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"
apply (unfold bounded_def)
apply clarify
apply (simp add: err_step_def split: err.splits)
apply (simp add: error_def)
 apply blast
apply (simp split: split_if_asm)
 apply (blast dest: in_map_sndD)
apply (simp add: error_def)
apply blast
done


lemma bounded_lift:
  "bounded step n ⟹ bounded (err_step n app step) n"
  apply (unfold bounded_def err_step_def error_def)
  apply clarify
  apply (erule allE, erule impE, assumption)
  apply (case_tac s)
  apply (auto simp add: map_snd_def split: split_if_asm)
  done


lemma le_list_map_OK [simp]:
  "⋀b. map OK a <=[Err.le r] map OK b = (a <=[r] b)"
  apply (induct a)
   apply simp
  apply simp
  apply (case_tac b)
   apply simp
  apply simp
  done


lemma map_snd_lessI:
  "x <=|r| y ⟹ map_snd OK x <=|Err.le r| map_snd OK y"
  apply (induct x)
  apply (unfold lesubstep_type_def map_snd_def)
  apply auto
  done
```

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)"
apply *(unfold app_mono_def mono_def err_step_def)*
apply *clarify*
apply *(case_tac s)*
apply *simp*
apply *simp*
apply *(case_tac t)*
apply *simp*
apply *clarify*
apply *(simp add: lesubstep_type_def error_def)*
apply *clarify*
apply *(drule in_map_sndD)*
apply *clarify*
apply *(drule bounded_err_stepD, assumption+)*
apply *(rule exI [of _ Err])*
apply *simp*
apply *simp*
apply *(erule allE, erule allE, erule allE, erule impE)*
apply *(rule conjI, assumption)*
apply *(rule conjI, assumption)*
apply *assumption*
apply *(rule conjI)*
apply *clarify*
apply *(erule allE, erule allE, erule allE, erule impE)*
apply *(rule conjI, assumption)*
apply *(rule conjI, assumption)*
apply *assumption*
apply *(erule impE, assumption)*
apply *(rule map_snd_lessI, assumption)*
apply *clarify*
apply *(simp add: lesubstep_type_def error_def)*
apply *clarify*
apply *(drule in_map_sndD)*
apply *clarify*
apply *(drule bounded_err_stepD, assumption+)*
apply *(rule exI [of _ Err])*
apply *simp*
done

lemma *in_errorD:*
"(x,y) ∈ set (error n) ⟹ y = Err"
by *(auto simp add: error_def)*

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)"
apply (unfold pres_type_def err_step_def)
apply clarify
apply (case_tac b)
 apply simp
apply (case_tac s)
 apply simp
 apply (drule in_errorD)
 apply simp
apply (simp add: map_snd_def split: split_if_asm)
 apply fast
apply (drule in_errorD)
apply simp
done
```

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 (unfold wt_app_eff_def, intro strip, rule conjI)
  fix p assume "p < size (map ok_val ts)"
  hence lp: "p < size ts" by simp
  hence ts: "0 < size ts" by (cases p) auto
  hence err: "(0,Err) ∈ set (error (size ts))" by (simp add: error_def)

  from wt lp
  have [intro?]: "⋀p. p < size ts ⟹ ts ! p ≠ Err"
    by (unfold wt_err_step_def wt_step_def) simp

  show app: "app p (map ok_val ts ! p)"
  proof (rule ccontr)
    from wt lp obtain s where
      OKp:  "ts ! p = OK s" and
      less: "∀(q,t) ∈ set (err_step (size ts) app step p (ts!p)). t <=_(Err.le r) ts!q"
      by (unfold wt_err_step_def wt_step_def stable_def)
         (auto iff: not_Err_eq)
    assume "¬ app p (map ok_val ts ! p)"
    with OKp lp have "¬ app p s" by simp
    with OKp have "err_step (size ts) app step p (ts!p) = error (size ts)"
      by (simp add: err_step_def)
    with err ts obtain q where
      "(q,Err) ∈ set (err_step (size ts) app step p (ts!p))" and
      q: "q < size ts" by auto
    with less have "ts!q = Err" by auto
    moreover from q have "ts!q ≠ Err" ..
    ultimately show False by blast
  qed

  show "∀(q,t)∈set(step p (map ok_val ts ! p)). t <=_r map ok_val ts ! q"
```

```
  proof clarify
    fix q t assume q: "(q,t) ∈ set (step p (map ok_val ts ! p))"

    from wt lp q
    obtain s where
      OKp:  "ts ! p = OK s" and
      less: "∀(q,t) ∈ set (err_step (size ts) app step p (ts!p)). t <=_(Err.le r) ts!q"
      by (unfold wt_err_step_def wt_step_def stable_def)
         (auto iff: not_Err_eq)

    from b lp app q have lq: "q < size ts" by (rule bounded_err_stepD)
    hence "ts!q ≠ Err" ..
    then obtain s' where OKq: "ts ! q = OK s'" by (auto iff: not_Err_eq)

    from lp lq OKp OKq app less q
    show "t <=_r map ok_val ts ! q"
      by (auto simp add: err_step_def map_snd_def)
  qed
qed


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 (unfold wt_err_step_def wt_step_def, intro strip, rule conjI)
  fix p assume "p < size (map OK ts)"
  hence p: "p < size ts" by simp
  thus "map OK ts ! p ≠ Err" by simp
  { fix q t
    assume q: "(q,t) ∈ set (err_step (size ts) app step p (map OK ts ! p))"
    with p app_eff obtain
      "app p (ts ! p)" "∀(q,t) ∈ set (step p (ts!p)). t <=_r ts!q"
      by (unfold wt_app_eff_def) blast
    moreover
    from q p bounded have "q < size ts"
      by - (rule boundedD)
    hence "map OK ts ! q = OK (ts!q)" by simp
    moreover
    have "p < size ts" by (rule p)
    moreover note q
    ultimately
    have "t <=_(Err.le r) map OK ts ! q"
      by (auto simp add: err_step_def map_snd_def)
  }
  thus "stable (Err.le r) (err_step (size ts) app step) (map OK ts) p"
    by (unfold stable_def) blast
qed

end
```

4.9 More about Options

theory *Opt = Err:*

constdefs
```
 le :: "'a ord ⇒ 'a option ord"
"le r o1 o2 == case o2 of None ⇒ o1=None |
                          Some y ⇒ (case o1 of None ⇒ True
                                             | Some x ⇒ x <=_r y)"

 opt :: "'a set ⇒ 'a option set"
"opt A == insert None {x . ? y:A. x = Some y}"

 sup :: "'a ebinop ⇒ 'a option ebinop"
"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)))"

 esl :: "'a esl ⇒ 'a option esl"
"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))"
```
apply *(unfold lesub_def le_def)*
apply *(rule refl)*
done

lemma *le_opt_refl:*
```
  "order r ⟹ o1 <=_(le r) o1"
```
by *(simp add: unfold_le_opt split: option.split)*

lemma *le_opt_trans [rule_format]:*
```
  "order r ⟹
   o1 <=_(le r) o2 ⟶ o2 <=_(le r) o3 ⟶ o1 <=_(le r) o3"
```
apply *(simp add: unfold_le_opt split: option.split)*
apply *(blast intro: order_trans)*
done

lemma *le_opt_antisym [rule_format]:*
```
  "order r ⟹ o1 <=_(le r) o2 ⟶ o2 <=_(le r) o1 ⟶ o1=o2"
```
apply *(simp add: unfold_le_opt split: option.split)*
apply *(blast intro: order_antisym)*
done

lemma *order_le_opt [intro!,simp]:*
```
  "order r ⟹ order(le r)"
```
apply *(subst order_def)*
apply *(blast intro: le_opt_refl le_opt_trans le_opt_antisym)*
done

lemma *None_bot [iff]:*
```
  "None <=_(le r) ox"
```

apply *(unfold lesub_def le_def)*
apply *(simp split: option.split)*
done

lemma *Some_le [iff]:*
  "(Some x <=_(le r) ox) = (? y. ox = Some y & x <=_r y)"
apply *(unfold lesub_def le_def)*
apply *(simp split: option.split)*
done

lemma *le_None [iff]:*
  "(ox <=_(le r) None) = (ox = None)"
apply *(unfold lesub_def le_def)*
apply *(simp split: option.split)*
done

lemma *OK_None_bot [iff]:*
  "OK None <=_(Err.le (le r)) x"
  by *(simp add: lesub_def Err.le_def le_def split: option.split err.split)*

lemma *sup_None1 [iff]:*
  "x +_(sup f) None = OK x"
  by *(simp add: plussub_def sup_def split: option.split)*

lemma *sup_None2 [iff]:*
  "None +_(sup f) x = OK x"
  by *(simp add: plussub_def sup_def split: option.split)*

lemma *None_in_opt [iff]:*
  "None : opt A"
by *(simp add: opt_def)*

lemma *Some_in_opt [iff]:*
  "(Some x : opt A) = (x:A)"
apply *(unfold opt_def)*
apply *auto*
done

lemma *semilat_opt [intro, simp]:*
  "⋀L. err_semilat L ⟹ err_semilat (Opt.esl L)"
proof *(unfold Opt.esl_def Err.sl_def, simp add: split_tupled_all)*

  fix *A r f*
  assume *s: "semilat (err A, Err.le r, lift2 f)"*

  let *?A0 = "err A"*
  let *?r0 = "Err.le r"*
  let *?f0 = "lift2 f"*

  from *s*
  obtain

```
    ord: "order ?r0" and
    clo: "closed ?A0 ?f0" and
    ub1: "∀x∈?A0. ∀y∈?A0. x <=_?r0 x +_?f0 y" and
    ub2: "∀x∈?A0. ∀y∈?A0. y <=_?r0 x +_?f0 y" and
    lub: "∀x∈?A0. ∀y∈?A0. ∀z∈?A0. x <=_?r0 z ∧ y <=_?r0 z ⟶ x +_?f0 y <=_?r0 z"
    by (unfold semilat_def) simp

  let ?A = "err (opt A)"
  let ?r = "Err.le (Opt.le r)"
  let ?f = "lift2 (Opt.sup f)"

  from ord
  have "order ?r"
    by simp

  moreover

  have "closed ?A ?f"
  proof (unfold closed_def, intro strip)
    fix x y
    assume x: "x : ?A"
    assume y: "y : ?A"

    { fix a b
      assume ab: "x = OK a" "y = OK b"

      with x
      have a: "⋀c. a = Some c ⟹ c : A"
        by (clarsimp simp add: opt_def)

      from ab y
      have b: "⋀d. b = Some d ⟹ d : A"
        by (clarsimp simp add: opt_def)

      { fix c d assume "a = Some c" "b = Some d"
        with ab x y
        have "c:A & d:A"
          by (simp add: err_def opt_def Bex_def)
        with clo
        have "f c d : err A"
          by (simp add: closed_def plussub_def err_def lift2_def)
        moreover
        fix z assume "f c d = OK z"
        ultimately
        have "z : A" by simp
      } note f_closed = this

      have "sup f a b : ?A"
      proof (cases a)
        case None
        thus ?thesis
          by (simp add: sup_def opt_def) (cases b, simp, simp add: b Bex_def)
      next
        case Some
```

```
      thus ?thesis
        by (auto simp add: sup_def opt_def Bex_def a b f_closed split: err.split option.split)
    qed
  }

  thus "x +_?f y : ?A"
    by (simp add: plussub_def lift2_def split: err.split)
qed

moreover

{ fix a b c
  assume "a ∈ opt A" "b ∈ opt A" "a +_(sup f) b = OK c"
  moreover
  from ord have "order r" by simp
  moreover
  { fix x y z
    assume "x ∈ A" "y ∈ A"
    hence "OK x ∈ err A ∧ OK y ∈ err A" by simp
    with ub1 ub2
    have "(OK x) <=_(Err.le r) (OK x) +_(lift2 f) (OK y) ∧
          (OK y) <=_(Err.le r) (OK x) +_(lift2 f) (OK y)"
      by blast
    moreover
    assume "x +_f y = OK z"
    ultimately
    have "x <=_r z ∧ y <=_r z"
      by (auto simp add: plussub_def lift2_def Err.le_def lesub_def)
  }
  ultimately
  have "a <=_(le r) c ∧ b <=_(le r) c"
    by (auto simp add: sup_def le_def lesub_def plussub_def
             dest: order_refl split: option.splits err.splits)
}

hence "(∀x∈?A. ∀y∈?A. x <=_?r x +_?f y) ∧ (∀x∈?A. ∀y∈?A. y <=_?r x +_?f y)"
  by (auto simp add: lesub_def plussub_def Err.le_def lift2_def split: err.split)

moreover

have "∀x∈?A. ∀y∈?A. ∀z∈?A. x <=_?r z ∧ y <=_?r z ⟶ x +_?f y <=_?r z"
proof (intro strip, elim conjE)
  fix x y z
  assume xyz: "x : ?A" "y : ?A" "z : ?A"
  assume xz: "x <=_?r z"
  assume yz: "y <=_?r z"

  { fix a b c
    assume ok: "x = OK a" "y = OK b" "z = OK c"

    { fix d e g
      assume some: "a = Some d" "b = Some e" "c = Some g"

      with ok xyz
```

```
        obtain "OK d:err A" "OK e:err A" "OK g:err A"
          by simp
        with lub
        have "⟦ (OK d) <=_(Err.le r) (OK g); (OK e) <=_(Err.le r) (OK g) ⟧
          ⟹ (OK d) +_(lift2 f) (OK e) <=_(Err.le r) (OK g)"
          by blast
        hence "⟦ d <=_r g; e <=_r g ⟧ ⟹ ∃y. d +_f e = OK y ∧ y <=_r g"
          by simp

        with ok some xyz xz yz
        have "x +_?f y <=_?r z"
          by (auto simp add: sup_def le_def lesub_def lift2_def plussub_def Err.le_def)
      } note this [intro!]

      from ok xyz xz yz
      have "x +_?f y <=_?r z"
        by - (cases a, simp, cases b, simp, cases c, simp, blast)
    }

    with xyz xz yz
    show "x +_?f y <=_?r z"
      by - (cases x, simp, cases y, simp, cases z, simp+)
  qed

  ultimately

  show "semilat (?A,?r,?f)"
    by (unfold semilat_def) simp
qed

lemma top_le_opt_Some [iff]:
  "top (le r) (Some T) = top r T"
apply (unfold top_def)
apply (rule iffI)
 apply blast
apply (rule allI)
apply (case_tac "x")
apply simp+
done

lemma Top_le_conv:
  "⟦ order r; top r T ⟧ ⟹ (T <=_r x) = (x = T)"
apply (unfold top_def)
apply (blast intro: order_antisym)
done


lemma acc_le_optI [intro!]:
  "acc r ⟹ acc(le r)"
apply (unfold acc_def lesub_def le_def lesssub_def)
apply (simp add: wf_eq_minimal split: option.split)
apply clarify
apply (case_tac "? a. Some a : Q")
 apply (erule_tac x = "{a . Some a : Q}" in allE)
```

```
 apply blast
apply (case_tac "x")
 apply blast
apply blast
done

lemma option_map_in_optionI:
  "⟦ ox : opt S; !x:S. ox = Some x ⟶ f x : S ⟧
  ⟹ option_map f ox : opt S"
apply (unfold option_map_def)
apply (simp split: option.split)
apply blast
done

end
```

4.10 The Java Type System as Semilattice

theory *JType = WellForm + Err:*

constdefs
```
  super :: "'a prog ⇒ cname ⇒ cname"
  "super G C == fst (the (class G C))"
```

lemma *superI:*
```
  "(C,D) ∈ subcls1 G ⟹ super G C = D"
  by (unfold super_def) (auto dest: subcls1D)
```

constdefs
```
  is_ref :: "ty ⇒ bool"
  "is_ref T == case T of PrimT t ⇒ False | RefT r ⇒ True"

  sup :: "'c prog ⇒ ty ⇒ ty ⇒ ty err"
  "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)))))"

  subtype :: "'c prog ⇒ ty ⇒ ty ⇒ bool"
  "subtype G T1 T2 == G ⊢ T1 ≼ T2"

  is_ty :: "'c prog ⇒ ty ⇒ bool"
  "is_ty G T == case T of PrimT P ⇒ True | RefT R ⇒
                (case R of NullT ⇒ True | ClassT C ⇒ (C,Object):(subcls1 G)^*)"
```

translations
```
  "types G" == "Collect (is_type G)"
```

constdefs
```
  esl :: "'c prog ⇒ ty esl"
  "esl G == (types G, subtype G, sup G)"
```

lemma *PrimT_PrimT:* `"(G ⊢ xb ≼ PrimT p) = (xb = PrimT p)"`
by *(auto elim: widen.elims)*

lemma *PrimT_PrimT2:* `"(G ⊢ PrimT p ≼ xb) = (xb = PrimT p)"`
by *(auto elim: widen.elims)*

lemma *is_tyI:*
```
  "⟦ is_type G T; wf_prog wf_mb G ⟧ ⟹ is_ty G T"
  by (auto simp add: is_ty_def intro: subcls_C_Object
           split: ty.splits ref_ty.splits)
```

lemma *is_type_conv:*
```
  "wf_prog wf_mb G ⟹ is_type G T = is_ty G T"
```
proof

```
  assume "is_type G T" "wf_prog wf_mb G"
  thus "is_ty G T"
    by (rule is_tyI)
next
  assume wf: "wf_prog wf_mb G" and
         ty: "is_ty G T"

  show "is_type G T"
  proof (cases T)
    case PrimT
    thus ?thesis by simp
  next
    fix R assume R: "T = RefT R"
    with wf
    have "R = ClassT Object ⟹ ?thesis" by simp
    moreover
    from R wf ty
    have "R ≠ ClassT Object ⟹ ?thesis"
     by (auto simp add: is_ty_def is_class_def split_tupled_all
               elim!: subcls1.elims
               elim: converse_rtranclE
               split: ref_ty.splits)
    ultimately
    show ?thesis by blast
  qed
qed

lemma order_widen:
  "acyclic (subcls1 G) ⟹ order (subtype G)"
  apply (unfold order_def lesub_def subtype_def)
  apply (auto intro: widen_trans)
  apply (case_tac x)
   apply (case_tac y)
    apply (auto simp add: PrimT_PrimT)
   apply (case_tac y)
    apply simp
  apply simp
  apply (case_tac ref_ty)
   apply (case_tac ref_tya)
    apply simp
   apply simp
  apply (case_tac ref_tya)
   apply simp
  apply simp
  apply (auto dest: acyclic_impl_antisym_rtrancl antisymD)
  done

lemma wf_converse_subcls1_impl_acc_subtype:
  "wf ((subcls1 G)^-1) ⟹ acc (subtype G)"
apply (unfold acc_def lesssub_def)
apply (drule_tac p = "(subcls1 G)^-1 - Id" in wf_subset)
 apply blast
apply (drule wf_trancl)
apply (simp add: wf_eq_minimal)
```

```
apply clarify
apply (unfold lesub_def subtype_def)
apply (rename_tac M T)
apply (case_tac "EX C. Class C : M")
 prefer 2
 apply (case_tac T)
  apply (fastsimp simp add: PrimT_PrimT2)
 apply simp
 apply (subgoal_tac "ref_ty = NullT")
  apply simp
  apply (rule_tac x = NT in bexI)
   apply (rule allI)
   apply (rule impI, erule conjE)
   apply (drule widen_RefT)
   apply clarsimp
   apply (case_tac t)
    apply simp
   apply simp
  apply simp
 apply (case_tac ref_ty)
  apply simp
 apply simp
apply (erule_tac x = "{C. Class C : M}" in allE)
apply auto
apply (rename_tac D)
apply (rule_tac x = "Class D" in bexI)
 prefer 2
 apply assumption
apply clarify
apply (frule widen_RefT)
apply (erule exE)
apply (case_tac t)
 apply simp
apply simp
apply (insert rtrancl_r_diff_Id [symmetric, standard, of "(subcls1 G)"])
apply simp
apply (erule rtranclE)
 apply blast
apply (drule rtrancl_converseI)
apply (subgoal_tac "((subcls1 G)-Id)^-1 = ((subcls1 G)^-1 - Id)")
 prefer 2
 apply blast
apply simp
apply (blast intro: rtrancl_into_trancl2)
done

lemma closed_err_types:
  "⟦ wf_prog wf_mb G; single_valued (subcls1 G); acyclic (subcls1 G) ⟧
  ⟹ closed (err (types G)) (lift2 (sup G))"
  apply (unfold closed_def plussub_def lift2_def sup_def)
  apply (auto split: err.split)
  apply (drule is_tyI, assumption)
  apply (auto simp add: is_ty_def is_type_conv simp del: is_type.simps
              split: ty.split ref_ty.split)
```

```
  apply (blast dest!: is_lub_exec_lub is_lubD is_ubD intro!: is_ubI superI)
  done


lemma sup_subtype_greater:
  "⟦ wf_prog wf_mb 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 -
  assume wf_prog:       "wf_prog wf_mb G"
  assume single_valued: "single_valued (subcls1 G)"
  assume acyclic:       "acyclic (subcls1 G)"

  { fix c1 c2
    assume is_class: "is_class G c1" "is_class G c2"
    with wf_prog
    obtain
      "G ⊢ c1 ≼C Object"
      "G ⊢ c2 ≼C Object"
      by (blast intro: subcls_C_Object)
    with wf_prog single_valued
    obtain u where
      "is_lub ((subcls1 G)^* ) c1 c2 u"
      by (blast dest: single_valued_has_lubs)
    moreover
    note acyclic
    moreover
    have "∀x y. (x, y) ∈ subcls1 G ⟶ super G x = y"
      by (blast intro: superI)
    ultimately
    have "G ⊢ c1 ≼C exec_lub (subcls1 G) (super G) c1 c2 ∧
          G ⊢ c2 ≼C exec_lub (subcls1 G) (super G) c1 c2"
      by (simp add: exec_lub_conv) (blast dest: is_lubD is_ubD)
  } note this [simp]

  assume "is_type G t1" "is_type G t2" "sup G t1 t2 = OK s"
  thus ?thesis
    apply (unfold sup_def subtype_def)
    apply (cases s)
    apply (auto split: ty.split_asm ref_ty.split_asm split_if_asm)
    done
qed

lemma sup_subtype_smallest:
  "⟦ wf_prog wf_mb 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 -
  assume wf_prog:       "wf_prog wf_mb G"
  assume single_valued: "single_valued (subcls1 G)"
  assume acyclic:       "acyclic (subcls1 G)"

  { fix c1 c2 D
```

```
    assume is_class: "is_class G c1" "is_class G c2"
    assume le: "G ⊢ c1 ≼C D" "G ⊢ c2 ≼C D"
    from wf_prog is_class
    obtain
      "G ⊢ c1 ≼C Object"
      "G ⊢ c2 ≼C Object"
      by (blast intro: subcls_C_Object)
    with wf_prog single_valued
    obtain u where
      lub: "is_lub ((subcls1 G)^* ) c1 c2 u"
      by (blast dest: single_valued_has_lubs)
    with acyclic
    have "exec_lub (subcls1 G) (super G) c1 c2 = u"
      by (blast intro: superI exec_lub_conv)
    moreover
    from lub le
    have "G ⊢ u ≼C D"
      by (simp add: is_lub_def is_ub_def)
    ultimately
    have "G ⊢ exec_lub (subcls1 G) (super G) c1 c2 ≼C D"
      by blast
  } note this [intro]

  have [dest!]:
    "⋀C T. G ⊢ Class C ≼ T ⟹ ∃D. T=Class D ∧ G ⊢ C ≼C D"
    by (frule widen_Class, auto)

  assume "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"
  thus ?thesis
    by (auto simp add: subtype_def sup_def
             split: ty.split_asm ref_ty.split_asm split_if_asm)
qed

lemma sup_exists:
  "⟦ subtype G a c; subtype G b c; sup G a b = Err ⟧ ⟹ False"
  by (auto simp add: PrimT_PrimT PrimT_PrimT2 sup_def subtype_def
           split: ty.splits ref_ty.splits)

lemma err_semilat_JType_esl_lemma:
  "⟦ wf_prog wf_mb G; single_valued (subcls1 G); acyclic (subcls1 G) ⟧
  ⟹ err_semilat (esl G)"
proof -
  assume wf_prog:   "wf_prog wf_mb G"
  assume single_valued: "single_valued (subcls1 G)"
  assume acyclic:   "acyclic (subcls1 G)"

  hence "order (subtype G)"
    by (rule order_widen)
  moreover
  from wf_prog single_valued acyclic
  have "closed (err (types G)) (lift2 (sup G))"
    by (rule closed_err_types)
  moreover
```

```
  from wf_prog single_valued acyclic
  have
    "(∀x∈err (types G). ∀y∈err (types G). x <=_(le (subtype G)) x +_(lift2 (sup G))
y) ∧
     (∀x∈err (types G). ∀y∈err (types G). y <=_(le (subtype G)) x +_(lift2 (sup G))
y)"
    by (auto simp add: lesub_def plussub_def Err.le_def lift2_def sup_subtype_greater
split: err.split)

  moreover

  from wf_prog single_valued acyclic
  have
    "∀x∈err (types G). ∀y∈err (types G). ∀z∈err (types G).
    x <=_(le (subtype G)) z ∧ y <=_(le (subtype G)) z ⟶ x +_(lift2 (sup G)) y <=_(le
(subtype G)) z"
    by (unfold lift2_def plussub_def lesub_def Err.le_def)
       (auto intro: sup_subtype_smallest sup_exists split: err.split)

  ultimately

  show ?thesis
    by (unfold esl_def semilat_def sl_def) auto
qed

lemma single_valued_subcls1:
  "wf_prog wf_mb G ⟹ single_valued (subcls1 G)"
  by (auto simp add: wf_prog_def unique_def single_valued_def
    intro: subcls1I elim!: subcls1.elims)

theorem err_semilat_JType_esl:
  "wf_prog wf_mb G ⟹ err_semilat (esl G)"
  by (frule acyclic_subcls1, frule single_valued_subcls1, rule err_semilat_JType_esl_lemma)

end
```

4.11 The JVM Type System as Semilattice

theory *JVMType = Opt + Product + Listn + JType:*

types
```
  locvars_type = "ty err list"
  opstack_type = "ty list"
  state_type   = "opstack_type × locvars_type"
  state        = "state_type option err"     — for Kildall
  method_type  = "state_type option list"    — for BVSpec
  class_type   = "sig ⇒ method_type"
  prog_type    = "cname ⇒ class_type"
```

constdefs
```
  stk_esl :: "'c prog ⇒ nat ⇒ ty list esl"
  "stk_esl S maxs == upto_esl maxs (JType.esl S)"

  reg_sl :: "'c prog ⇒ nat ⇒ ty err list sl"
  "reg_sl S maxr == Listn.sl maxr (Err.sl (JType.esl S))"

  sl :: "'c prog ⇒ nat ⇒ nat ⇒ state sl"
  "sl S maxs maxr ==
  Err.sl(Opt.esl(Product.esl (stk_esl S maxs) (Err.esl(reg_sl S maxr))))"
```

constdefs
```
  states :: "'c prog ⇒ nat ⇒ nat ⇒ state set"
  "states S maxs maxr == fst(sl S maxs maxr)"

  le :: "'c prog ⇒ nat ⇒ nat ⇒ state ord"
  "le S maxs maxr == fst(snd(sl S maxs maxr))"

  sup :: "'c prog ⇒ nat ⇒ nat ⇒ state binop"
  "sup S maxs maxr == snd(snd(sl S maxs maxr))"
```

constdefs
```
  sup_ty_opt :: "['code prog,ty err,ty err] ⇒ bool"
                 ("_ |- _ <=o _" [71,71] 70)
  "sup_ty_opt G == Err.le (subtype G)"

  sup_loc :: "['code prog,locvars_type,locvars_type] ⇒ bool"
              ("_ |- _ <=l _"  [71,71] 70)
  "sup_loc G == Listn.le (sup_ty_opt G)"

  sup_state :: "['code prog,state_type,state_type] ⇒ bool"
               ("_ |- _ <=s _"  [71,71] 70)
  "sup_state G == Product.le (Listn.le (subtype G)) (sup_loc G)"

  sup_state_opt :: "['code prog,state_type option,state_type option] ⇒ bool"
                   ("_ |- _ <=' _"  [71,71] 70)
  "sup_state_opt G == Opt.le (sup_state G)"
```

syntax *(xsymbols)*
```
  sup_ty_opt    :: "['code prog,ty err,ty err] ⇒ bool"
                   ("_ ⊢ _ <=o _" [71,71] 70)
  sup_loc       :: "['code prog,locvars_type,locvars_type] ⇒ bool"
                   ("_ ⊢ _ <=l _" [71,71] 70)
  sup_state     :: "['code prog,state_type,state_type] ⇒ bool"
                   ("_ ⊢ _ <=s _" [71,71] 70)
  sup_state_opt :: "['code prog,state_type option,state_type option] ⇒ bool"
                   ("_ ⊢ _ <=' _" [71,71] 70)
```

lemma *JVM_states_unfold:*
```
  "states S maxs maxr == err(opt((Union {list n (types S) |n. n <= maxs}) <*>
                                 list maxr (err(types S))))"
  apply (unfold states_def sl_def Opt.esl_def Err.sl_def
         stk_esl_def reg_sl_def Product.esl_def
         Listn.sl_def upto_esl_def JType.esl_def Err.esl_def)
  by simp
```

lemma *JVM_le_unfold:*
```
 "le S m n ==
  Err.le(Opt.le(Product.le(Listn.le(subtype S))(Listn.le(Err.le(subtype S)))))"
  apply (unfold le_def sl_def Opt.esl_def Err.sl_def
         stk_esl_def reg_sl_def Product.esl_def
         Listn.sl_def upto_esl_def JType.esl_def Err.esl_def)
  by simp
```

lemma *JVM_le_convert:*
```
  "le G m n (OK t1) (OK t2) = G ⊢ t1 <=' t2"
  by (simp add: JVM_le_unfold Err.le_def lesub_def sup_state_opt_def
                sup_state_def sup_loc_def sup_ty_opt_def)
```

lemma *JVM_le_Err_conv:*
```
  "le G m n = Err.le (sup_state_opt G)"
  by (unfold sup_state_opt_def sup_state_def sup_loc_def
             sup_ty_opt_def JVM_le_unfold) simp
```

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)"
  apply (induct b)
   apply simp
  apply clarsimp
  apply (case_tac aa)
  apply simp+
  done
```

lemma *[simp]: "Err.le r (OK a) (OK b) = r a b"*
```
  by (simp add: Err.le_def lesub_def)
```

lemma *stk_convert:*
```
  "Listn.le (subtype G) a b = G ⊢ map OK a <=l map OK b"
```
proof

```
  assume "Listn.le (subtype G) a b"

  hence le: "list_all2 (subtype G) a b"
    by (unfold Listn.le_def lesub_def)

  { fix x' y'
    assume "length a = length b"
           "(x',y') ∈ set (zip (map OK a) (map OK b))"
    then
    obtain x y where OK:
      "x' = OK x" "y' = OK y" "(x,y) ∈ set (zip a b)"
      by (auto simp add: zip_map)
    with le
    have "subtype G x y"
      by (simp add: list_all2_def Ball_def)
    with OK
    have "G ⊢ x' <=o y'"
      by (simp add: sup_ty_opt_def)
  }

  with le
  show "G ⊢ map OK a <=l map OK b"
    by (unfold sup_loc_def Listn.le_def lesub_def list_all2_def) auto
next
  assume "G ⊢ map OK a <=l map OK b"

  thus "Listn.le (subtype G) a b"
    apply (unfold sup_loc_def list_all2_def Listn.le_def lesub_def)
    apply (clarsimp simp add: zip_map)
    apply (drule bspec, assumption)
    apply (auto simp add: sup_ty_opt_def subtype_def)
    done
qed


lemma sup_state_conv:
  "(G ⊢ s1 <=s s2) ==
  (G ⊢ map OK (fst s1) <=l map OK (fst s2)) ∧ (G ⊢ snd s1 <=l snd s2)"
  by (auto simp add: sup_state_def stk_convert lesub_def Product.le_def split_beta)


lemma subtype_refl [simp]:
  "subtype G t t"
  by (simp add: subtype_def)

theorem sup_ty_opt_refl [simp]:
  "G ⊢ t <=o t"
  by (simp add: sup_ty_opt_def Err.le_def lesub_def split: err.split)

lemma le_list_refl2 [simp]:
  "(⋀xs. r xs xs) ⟹ Listn.le r xs xs"
  by (induct xs, auto simp add: Listn.le_def lesub_def)

theorem sup_loc_refl [simp]:
```

```
  "G ⊢ t <=l t"
  by (simp add: sup_loc_def)

theorem sup_state_refl [simp]:
  "G ⊢ s <=s s"
  by (auto simp add: sup_state_def Product.le_def lesub_def)

theorem sup_state_opt_refl [simp]:
  "G ⊢ s <=' s"
  by (simp add: sup_state_opt_def Opt.le_def lesub_def split: option.split)


theorem anyConvErr [simp]:
  "(G ⊢ Err <=o any) = (any = Err)"
  by (simp add: sup_ty_opt_def Err.le_def split: err.split)

theorem OKanyConvOK [simp]:
  "(G ⊢ (OK ty') <=o (OK ty)) = (G ⊢ ty' ≼ ty)"
  by (simp add: sup_ty_opt_def Err.le_def lesub_def subtype_def)

theorem sup_ty_opt_OK:
  "G ⊢ a <=o (OK b) ⟹ ∃ x. a = OK x"
  by (clarsimp simp add: sup_ty_opt_def Err.le_def split: err.splits)

lemma widen_PrimT_conv1 [simp]:
  "⟦ G ⊢ S ≼ T; S = PrimT x⟧ ⟹ T = PrimT x"
  by (auto elim: widen.elims)

theorem sup_PTS_eq:
  "(G ⊢ OK (PrimT p) <=o X) = (X=Err ∨ X = OK (PrimT p))"
  by (auto simp add: sup_ty_opt_def Err.le_def lesub_def subtype_def
             split: err.splits)

theorem sup_loc_Nil [iff]:
  "(G ⊢ [] <=l XT) = (XT=[])"
  by (simp add: sup_loc_def Listn.le_def)

theorem sup_loc_Cons [iff]:
  "(G ⊢ (Y#YT) <=l XT) = (∃X XT'. XT=X#XT' ∧ (G ⊢ Y <=o X) ∧ (G ⊢ YT <=l XT'))"
  by (simp add: sup_loc_def Listn.le_def lesub_def list_all2_Cons1)

theorem sup_loc_Cons2:
  "(G ⊢ YT <=l (X#XT)) = (∃Y YT'. YT=Y#YT' ∧ (G ⊢ Y <=o X) ∧ (G ⊢ YT' <=l XT))"
  by (simp add: sup_loc_def Listn.le_def lesub_def list_all2_Cons2)

lemma sup_state_Cons:
  "(G ⊢ (x#xt, a) <=s (y#yt, b)) =
   ((G ⊢ x ≼ y) ∧ (G ⊢ (xt,a) <=s (yt,b)))"
  by (auto simp add: sup_state_def stk_convert lesub_def Product.le_def)


theorem sup_loc_length:
  "G ⊢ a <=l b ⟹ length a = length b"
proof -
```

```
  assume G: "G ⊢ a <=l b"
  have "∀b. (G ⊢ a <=l b) ⟶ length a = length b"
    by (induct a, auto)
  with G
  show ?thesis by blast
qed

theorem sup_loc_nth:
  "⟦ G ⊢ a <=l b; n < length a ⟧ ⟹ G ⊢ (a!n) <=o (b!n)"
proof -
  assume a: "G ⊢ a <=l b" "n < length a"
  have "∀ n b. (G ⊢ a <=l b) ⟶ n < length a ⟶ (G ⊢ (a!n) <=o (b!n))"
    (is "?P a")
  proof (induct a)
    show "?P []" by simp

    fix x xs assume IH: "?P xs"

    show "?P (x#xs)"
    proof (intro strip)
      fix n b
      assume "G ⊢ (x # xs) <=l b" "n < length (x # xs)"
      with IH
      show "G ⊢ ((x # xs) ! n) <=o (b ! n)"
        by - (cases n, auto)
    qed
  qed
  with a
  show ?thesis by blast
qed

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 (induct a)
  show "?P []" by simp

  fix l ls assume IH: "?P ls"

  show "?P (l#ls)"
  proof (intro strip)
    fix b
    assume f: "∀n. n < length (l # ls) ⟶ (G ⊢ ((l # ls) ! n) <=o (b ! n))"
    assume l: "length (l#ls) = length b"

    then obtain b' bs where b: "b = b'#bs"
      by - (cases b, simp, simp add: neq_Nil_conv, rule that)

    with f
    have "∀n. n < length ls ⟶ (G ⊢ (ls!n) <=o (bs!n))"
      by auto

    with f b l IH
    show "G ⊢ (l # ls) <=l b"
```

```
      by auto
  qed
qed


theorem sup_loc_append:
  "length a = length b ⟹
   (G ⊢ (a@x) <=l (b@y)) = ((G ⊢ a <=l b) ∧ (G ⊢ x <=l y))"
proof -
  assume l: "length a = length b"

  have "∀b. length a = length b ⟶ (G ⊢ (a@x) <=l (b@y)) = ((G ⊢ a <=l b) ∧
           (G ⊢ x <=l y))" (is "?P a")
  proof (induct a)
    show "?P []" by simp

    fix l ls assume IH: "?P ls"
    show "?P (l#ls)"
    proof (intro strip)
      fix b
      assume "length (l#ls) = length (b::ty err list)"
      with IH
      show "(G ⊢ ((l#ls)@x) <=l (b@y)) = ((G ⊢ (l#ls) <=l b) ∧ (G ⊢ x <=l y))"
        by - (cases b, auto)
    qed
  qed
  with l
  show ?thesis by blast
qed

theorem sup_loc_rev [simp]:
  "(G ⊢ (rev a) <=l rev b) = (G ⊢ a <=l b)"
proof -
  have "∀b. (G ⊢ (rev a) <=l rev b) = (G ⊢ a <=l b)" (is "∀b. ?Q a b" is "?P a")
  proof (induct a)
    show "?P []" by simp

    fix l ls assume IH: "?P ls"

    {
      fix b
      have "?Q (l#ls) b"
      proof (cases (open) b)
        case Nil
        thus ?thesis by (auto dest: sup_loc_length)
      next
        case Cons
        show ?thesis
        proof
          assume "G ⊢ (l # ls) <=l b"
          thus "G ⊢ rev (l # ls) <=l rev b"
            by (clarsimp simp add: Cons IH sup_loc_length sup_loc_append)
        next
          assume "G ⊢ rev (l # ls) <=l rev b"
```

```
      hence G: "G ⊢ (rev ls @ [l]) <=l (rev list @ [a])"
        by (simp add: Cons)

      hence "length (rev ls) = length (rev list)"
        by (auto dest: sup_loc_length)

      from this G
      obtain "G ⊢ rev ls <=l rev list" "G ⊢ l <=o a"
        by (simp add: sup_loc_append)

      thus "G ⊢ (l # ls) <=l b"
        by (simp add: Cons IH)
    qed
  qed
 }
 thus "?P (l#ls)" by blast
qed

thus ?thesis by blast
qed


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 (induct x)
 show "?P []" by simp

 fix l ls assume IH: "?P ls"
 show "?P (l#ls)"
 proof (intro strip)
  fix n y
  assume "G ⊢a <=o b" "G ⊢ (l # ls) <=l y" "n < length y"
  with IH
  show "G ⊢ (l # ls)[n := a] <=l y[n := b]"
   by - (cases n, auto simp add: sup_loc_Cons2 list_all2_Cons1)
 qed
qed


theorem sup_state_length [simp]:
 "G ⊢ s2 <=s s1 ⟹
  length (fst s2) = length (fst s1) ∧ length (snd s2) = length (snd s1)"
 by (auto dest: sup_loc_length simp add: sup_state_def stk_convert lesub_def Product.le_def)

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)))"
 by (auto simp add: sup_state_def stk_convert lesub_def Product.le_def sup_loc_append)

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)))"
 by (auto simp add: sup_state_def stk_convert lesub_def Product.le_def sup_loc_append)
```

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)))"
```
by *(auto simp add: sup_state_def stk_convert lesub_def Product.le_def map_eq_Cons)*

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)))"
```
by *(auto simp add: sup_state_def stk_convert lesub_def Product.le_def map_eq_Cons sup_loc_Cons2)*

theorem *sup_state_ignore_fst:*
```
"G ⊢ (a, x) <=s (b, y) ⟹ G ⊢ (c, x) <=s (c, y)"
```
by *(simp add: sup_state_def lesub_def Product.le_def)*

theorem *sup_state_rev_fst:*
```
"(G ⊢ (rev a, x) <=s (rev b, y)) = (G ⊢ (a, x) <=s (b, y))"
```
proof -
have *m:* `"⋀f x. map f (rev x) = rev (map f x)"` **by** *(simp add: rev_map)*
show *?thesis* **by** *(simp add: m sup_state_def stk_convert lesub_def Product.le_def)*
qed

lemma *sup_state_opt_None_any [iff]:*
```
"(G ⊢ None <=' any) = True"
```
by *(simp add: sup_state_opt_def Opt.le_def split: option.split)*

lemma *sup_state_opt_any_None [iff]:*
```
"(G ⊢ any <=' None) = (any = None)"
```
by *(simp add: sup_state_opt_def Opt.le_def split: option.split)*

lemma *sup_state_opt_Some_Some [iff]:*
```
"(G ⊢ (Some a) <=' (Some b)) = (G ⊢ a <=s b)"
```
by *(simp add: sup_state_opt_def Opt.le_def lesub_def del: split_paired_Ex)*

lemma *sup_state_opt_any_Some [iff]:*
```
"(G ⊢ (Some a) <=' any) = (∃b. any = Some b ∧ G ⊢ a <=s b)"
```
by *(simp add: sup_state_opt_def Opt.le_def lesub_def split: option.split)*

lemma *sup_state_opt_Some_any:*
```
"(G ⊢ any <=' (Some b)) = (any = None ∨ (∃a. any = Some a ∧ G ⊢ a <=s b))"
```
by *(simp add: sup_state_opt_def Opt.le_def lesub_def split: option.split)*

theorem *sup_ty_opt_trans [trans]:*
```
"⟦G ⊢ a <=o b; G ⊢ b <=o c⟧ ⟹ G ⊢ a <=o c"
```
by *(auto intro: widen_trans*
simp add: sup_ty_opt_def Err.le_def lesub_def subtype_def
split: err.splits)

theorem *sup_loc_trans [trans]:*
```
"⟦G ⊢ a <=l b; G ⊢ b <=l c⟧ ⟹ G ⊢ a <=l c"
```
proof -
assume *G:* `"G ⊢ a <=l b" "G ⊢ b <=l c"`

```
  hence "∀ n. n < length a ⟶ (G ⊢ (a!n) <=o (c!n))"
  proof (intro strip)
    fix n
    assume n: "n < length a"
    with G
    have "G ⊢ (a!n) <=o (b!n)"
      by - (rule sup_loc_nth)
    also
    from n G
    have "G ⊢ ... <=o (c!n)"
      by - (rule sup_loc_nth, auto dest: sup_loc_length)
    finally
    show "G ⊢ (a!n) <=o (c!n)" .
  qed

  with G
  show ?thesis
    by (auto intro!: all_nth_sup_loc [rule_format] dest!: sup_loc_length)
qed


theorem sup_state_trans [trans]:
  "⟦G ⊢ a <=s b; G ⊢ b <=s c⟧ ⟹ G ⊢ a <=s c"
  by (auto intro: sup_loc_trans simp add: sup_state_def stk_convert Product.le_def lesub_def)

theorem sup_state_opt_trans [trans]:
  "⟦G ⊢ a <=' b; G ⊢ b <=' c⟧ ⟹ G ⊢ a <=' c"
  by (auto intro: sup_state_trans
           simp add: sup_state_opt_def Opt.le_def lesub_def
           split: option.splits)

end
```

4.12 Effect of Instructions on the State Type

theory *Effect = JVMType + JVMExceptions:*

types
succ_type = "(p_count × state_type option) list"

Program counter of successor instructions:

consts
succs :: "instr ⇒ p_count ⇒ p_count list"
primrec
```
"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:

consts
eff' :: "instr × jvm_prog × state_type ⇒ state_type"

recdef *eff' "{}"*
```
"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))"
```

consts
```
  match_any :: "jvm_prog ⇒ p_count ⇒ exception_table ⇒ cname list"
```
primrec
```
  "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')"
```

consts
```
  match :: "jvm_prog ⇒ xcpt ⇒ p_count ⇒ exception_table ⇒ cname list"
```
primrec
```
  "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 [])"
```
  by *(induct et) auto*

consts
```
  xcpt_names :: "instr × jvm_prog × p_count × exception_table ⇒ cname list"
```
recdef *xcpt_names "{}"*
```
  "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)            = []"
```

constdefs
```
  xcpt_eff :: "instr ⇒ jvm_prog ⇒ p_count ⇒ state_type option ⇒ exception_table ⇒
succ_type"
  "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))"

  norm_eff :: "instr ⇒ jvm_prog ⇒ state_type option ⇒ state_type option"
  "norm_eff i G == option_map (λs. eff' (i,G,s))"

  eff :: "instr ⇒ jvm_prog ⇒ p_count ⇒ exception_table ⇒ state_type option ⇒ succ_type"
  "eff i G pc et s == (map (λpc'. (pc',norm_eff i G s)) (succs i pc)) @ (xcpt_eff i G
pc s et)"
```

constdefs

```
  isPrimT :: "ty ⇒ bool"
  "isPrimT T == case T of PrimT T' ⇒ True | RefT T' ⇒ False"

  isRefT :: "ty ⇒ bool"
  "isRefT T == case T of PrimT T' ⇒ False | RefT T' ⇒ True"
```

lemma *isPrimT [simp]:*

```
  "isPrimT T = (∃T'. T = PrimT T')"
```

by *(simp add: isPrimT_def split: ty.splits)*

lemma *isRefT [simp]:*

```
  "isRefT T = (∃T'. T = RefT T')"
```

by *(simp add: isRefT_def split: ty.splits)*

lemma *"list_all2 P a b ⟹ ∀(x,y) ∈ set (zip a b). P x y"*

by *(simp add: list_all2_def)*

Conditions under which eff is applicable:

consts

```
app' :: "instr × jvm_prog × p_count × nat × ty × state_type ⇒ bool"
```

recdef *app' "{}"*

```
"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"
```

constdefs

```
  xcpt_app :: "instr ⇒ jvm_prog ⇒ nat ⇒ exception_table ⇒ bool"
  "xcpt_app i G pc et ≡ ∀C∈set(xcpt_names (i,G,pc,et)). is_class G C"

  app :: "instr ⇒ jvm_prog ⇒ nat ⇒ ty ⇒ nat ⇒ exception_table ⇒ state_type option
⇒ bool"
  "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"
```

apply *(induct et)*

apply *simp*

apply *simp*

apply *clarify*

apply *(simp split: split_if_asm)*

```
  apply (auto simp add: match_exception_entry_def)
  done

lemma match_X_match_table:
  "C ∈ set (match G X pc et) ⟹ match_exception_table G C pc et ≠ None"
  apply (induct et)
   apply simp
  apply (simp split: split_if_asm)
  done

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))"
  apply (cases i)
  apply (auto dest!: match_any_match_table match_X_match_table
              dest: match_exception_table_in_et)
  done


lemma 1: "2 < length a ⟹ (∃l l' l'' ls. a = l#l'#l''#ls)"
proof (cases a)
  fix x xs assume "a = x#xs" "2 < length a"
  thus ?thesis by - (cases xs, simp, cases "tl xs", auto)
qed auto

lemma 2: "¬(2 < length a) ⟹ a = [] ∨ (∃ l. a = [l]) ∨ (∃ l l'. a = [l,l'])"
proof -
  assume "¬(2 < length a)"
  hence "length a < (Suc (Suc (Suc 0)))" by simp
  hence * : "length a = 0 ∨ length a = Suc 0 ∨ length a = Suc (Suc 0)"
    by (auto simp add: less_Suc_eq)

  {
    fix x
    assume "length x = Suc 0"
    hence "∃ l. x = [l]"  by - (cases x, auto)
  } note 0 = this

  have "length a = Suc (Suc 0) ⟹ ∃l l'. a = [l,l']" by (cases a, auto dest: 0)
  with * show ?thesis by (auto dest: 0)
qed

lemmas [simp] = app_def xcpt_app_def
```

simp rules for `app`

```
lemma appNone[simp]: "app i G maxs rT pc et None = True" by simp
```

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)"
by *(cases s, simp)*

lemma *appStore[simp]:*
"(app (Store idx) G maxs rT pc et (Some s)) = (∃ts ST LT. s = (ts#ST,LT) ∧ idx < length LT)"
by *(cases s, cases "2 < length (fst s)", auto dest: 1 2)*

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)"
by *(cases s, simp)*

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))"
by *(cases s, cases "2 <length (fst s)", auto dest!: 1 2)*

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))"
by *(cases s, cases "2 <length (fst s)", auto dest!: 1 2)*

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))"
by *(cases s, simp)*

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))"
by *(cases s, cases "fst s", simp add: app_def) (cases "hd (fst s)", auto)*

lemma *appPop[simp]:*
"(app Pop G maxs rT pc et (Some s)) = (∃ts ST LT. s = (ts#ST,LT))"
by *(cases s, cases "2 <length (fst s)", auto dest: 1 2)*

lemma *appDup[simp]:*
"(app Dup G maxs rT pc et (Some s)) = (∃ts ST LT. s = (ts#ST,LT) ∧ 1+length ST < maxs)"

```
  by (cases s, cases "2 <length (fst s)", auto dest: 1 2)


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)"
  by (cases s, cases "2 <length (fst s)", auto dest: 1 2)


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)"
  by (cases s, cases "2 <length (fst s)", auto dest: 1 2)


lemma appSwap[simp]:
"app Swap G maxs rT pc et (Some s) = (∃ts1 ts2 ST LT. s = (ts1#ts2#ST,LT))"
  by (cases s, cases "2 <length (fst s)", auto dest: 1 2)


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 (cases (open) s)
  case Pair
  have "?app (a,b) = ?P (a,b)"
  proof (cases "a")
    fix t ts assume a: "a = t#ts"
    show ?thesis
    proof (cases t)
      fix p assume p: "t = PrimT p"
      show ?thesis
      proof (cases p)
        assume ip: "p = Integer"
        show ?thesis
        proof (cases ts)
          fix t' ts' assume t': "ts = t' # ts'"
          show ?thesis
          proof (cases t')
            fix p' assume "t' = PrimT p'"
            with t' ip p a
            show ?thesis by - (cases p', auto)
          qed (auto simp add: a p ip t')
        qed (auto simp add: a p ip)
      qed (auto simp add: a p)
    qed (auto simp add: a)
  qed auto
```

```
  with Pair show ?thesis by simp
qed


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')))"
  by (cases s, cases "2 <length (fst s)", auto dest!: 1 2)

lemma appReturn[simp]:
"app Return G maxs rT pc et (Some s) = (∃T ST LT. s = (T#ST,LT) ∧ (G ⊢ T ≼ rT))"
  by (cases s, cases "2 <length (fst s)", auto dest: 1 2)

lemma appGoto[simp]:
"app (Goto b) G maxs rT pc et (Some s) = (0 ≤ int pc + b)"
  by simp

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))"
  by (cases s, cases "2 < length (fst s)", auto dest: 1 2)

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 (cases (open) s)
  note list_all2_def [simp]
  case Pair
  have "?app (a,b) ⟹ ?P (a,b)"
  proof -
    assume app: "?app (a,b)"
    hence "a = (rev (rev (take (length fpTs) a))) @ (drop (length fpTs) a) ∧
          length fpTs < length a" (is "?a ∧ ?l")
      by (auto simp add: app_def)
    hence "?a ∧ 0 < length (drop (length fpTs) a)" (is "?a ∧ ?l")
      by auto
    hence "?a ∧ ?l ∧ length (rev (take (length fpTs) a)) = length fpTs"
      by (auto simp add: min_def)
    hence "∃apTs ST. a = rev apTs @ ST ∧ length apTs = length fpTs ∧ 0 < length ST"

      by blast
    hence "∃apTs ST. a = rev apTs @ ST ∧ length apTs = length fpTs ∧ ST ≠ []"
      by blast
```

```
    hence "∃apTs ST. a = rev apTs @ ST ∧ length apTs = length fpTs ∧
           (∃X ST'. ST = X#ST')"
      by (simp add: neq_Nil_conv)
    hence "∃apTs X ST. a = rev apTs @ X # ST ∧ length apTs = length fpTs"
      by blast
    with app
    show ?thesis by (unfold app_def, clarsimp) blast
  qed
  with Pair
  have "?app s ⟹ ?P s" by (simp only:)
  moreover
  have "?P s ⟹ ?app s" by (unfold app_def) (clarsimp simp add: min_def)
  ultimately
  show ?thesis by (rule iffI)
qed

lemma effNone:
  "(pc', s') ∈ set (eff i G pc et None) ⟹ s' = None"
  by (auto simp add: eff_def xcpt_eff_def norm_eff_def)
```

some helpers to make the specification directly executable:

```
declare list_all2_Nil [code]
declare list_all2_Cons [code]

lemma xcpt_app_lemma [code]:
  "xcpt_app i G pc et = list_all (is_class G) (xcpt_names (i, G, pc, et))"
  by (simp add: list_all_conv)

lemmas [simp del] = app_def xcpt_app_def

end
```

4.13 Monotonicity of eff and app

theory *EffectMono = Effect:*

lemma *PrimT_PrimT: "(G ⊢ xb ≼ PrimT p) = (xb = PrimT p)"*
by *(auto elim: widen.elims)*

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)))" (***is** *"?P b")*
proof *(induct (***open***) ?P b)*
show *"?P []"* **by** *simp*

case *Cons*
show *"?P (a#list)"*
proof *(clarsimp simp add: list_all2_Cons1 sup_loc_def Listn.le_def lesub_def)*
fix *z zs n*
assume ** :*
"G ⊢ a <=o z" "list_all2 (sup_ty_opt G) list zs"
"n < Suc (length list)" "(z # zs) ! n = OK t"

show *"(∃t. (a # list) ! n = OK t) ∧ G ⊢(a # list) ! n <=o OK t"*
proof *(cases n)*
case *0*
with *** **show** *?thesis* **by** *(simp add: sup_ty_opt_OK)*
next
case *Suc*
with *Cons **
show *?thesis* **by** *(simp add: sup_loc_def Listn.le_def lesub_def)*
qed
qed
qed

lemma *all_widen_is_sup_loc:*
"∀b. length a = length b ⟶
(∀x∈set (zip a b). x ∈ widen G) = (G ⊢ (map OK a) <=l (map OK b))"
*(***is** *"∀b. length a = length b ⟶ ?Q a b"* **is** *"?P a")*
proof *(induct "a")*
show *"?P []"* **by** *simp*

fix *l ls* **assume** *Cons: "?P ls"*

show *"?P (l#ls)"*
proof *(intro allI impI)*
fix *b*
assume *"length (l # ls) = length (b::ty list)"*
with *Cons*
show *"?Q (l # ls) b"* **by** *- (cases b, auto)*
qed
qed

```
lemma append_length_n [rule_format]:
"∀n. n ≤ length x ⟶ (∃a b. x = a@b ∧ length a = n)" (is "?P x")
proof (induct (open) ?P x)
  show "?P []" by simp

  fix l ls assume Cons: "?P ls"

  show "?P (l#ls)"
  proof (intro allI impI)
    fix n
    assume l: "n ≤ length (l # ls)"

    show "∃a b. l # ls = a @ b ∧ length a = n"
    proof (cases n)
      assume "n=0" thus ?thesis by simp
    next
      fix n' assume s: "n = Suc n'"
      with l have  "n' ≤ length ls" by simp
      hence "∃a b. ls = a @ b ∧ length a = n'" by (rule Cons [rule_format])
      then obtain a b where "ls = a @ b" "length a = n'" by rules
      with s have "l # ls = (l#a) @ b ∧ length (l#a) = n" by simp
      thus ?thesis by blast
    qed
  qed
qed

lemma rev_append_cons:
"n < length x ⟹ ∃a b c. x = (rev a) @ b # c ∧ length a = n"
proof -
  assume n: "n < length x"
  hence "n ≤ length x" by simp
  hence "∃a b. x = a @ b ∧ length a = n" by (rule append_length_n)
  then obtain r d where x: "x = r@d" "length r = n" by rules
  with n have "∃b c. d = b#c" by (simp add: neq_Nil_conv)
  then obtain b c where "d = b#c" by rules
  with x have "x = (rev (rev r)) @ b # c ∧ length (rev r) = n" by simp
  thus ?thesis by blast
qed

lemma sup_loc_length_map:
  "G ⊢ map f a <=l map g b ⟹ length a = length b"
proof -
  assume "G ⊢ map f a <=l map g b"
  hence "length (map f a) = length (map g b)" by (rule sup_loc_length)
  thus ?thesis by simp
qed

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 -
```

```
{ fix s1 s2
  assume G:   "G ⊢ s2 <=s s1"
  assume app: "app i G m rT pc et (Some s1)"

  note [simp] = sup_loc_length sup_loc_length_map

  have "app i G m rT pc et (Some s2)"
  proof (cases (open) i)
    case Load

    from G Load app
    have "G ⊢ snd s2 <=l snd s1" by (auto simp add: sup_state_conv)

    with G Load app show ?thesis
      by (cases s2) (auto simp add: sup_state_conv dest: sup_loc_some)
  next
    case Store
    with G app show ?thesis
      by (cases s2, auto simp add: map_eq_Cons sup_loc_Cons2 sup_state_conv)
  next
    case LitPush
    with G app show ?thesis by (cases s2, auto simp add: sup_state_conv)
  next
    case New
    with G app show ?thesis by (cases s2, auto simp add: sup_state_conv)
  next
    case Getfield
    with app G show ?thesis
      by (cases s2) (clarsimp simp add: sup_state_Cons2, rule widen_trans)
  next
    case Putfield

    with app
    obtain vT oT ST LT b
      where s1: "s1 = (vT # oT # ST, LT)" and
                "field (G, cname) vname = Some (cname, b)"
                "is_class G cname" and
            oT: "G⊢ oT≼ (Class cname)" and
            vT: "G⊢ vT≼ b" and
            xc: "Ball (set (match G NullPointer pc et)) (is_class G)"
      by force
    moreover
    from s1 G
    obtain vT' oT' ST' LT'
      where s2:  "s2 = (vT' # oT' # ST', LT')" and
            oT': "G⊢ oT' ≼ oT" and
            vT': "G⊢ vT' ≼ vT"
      by - (cases s2, simp add: sup_state_Cons2, elim exE conjE, simp, rule that)
    moreover
    from vT' vT
    have "G ⊢ vT' ≼ b" by (rule widen_trans)
    moreover
    from oT' oT
    have "G⊢ oT' ≼ (Class cname)" by (rule widen_trans)
```

```
    ultimately
    show ?thesis by (auto simp add: Putfield xc)
  next
    case Checkcast
    with app G show ?thesis
      by (cases s2, auto intro!: widen_RefT2 simp add: sup_state_Cons2)
  next
    case Return
    with app G show ?thesis
      by (cases s2) (auto simp add: sup_state_Cons2, rule widen_trans)
  next
    case Pop
    with app G show ?thesis
      by (cases s2, clarsimp simp add: sup_state_Cons2)
  next
    case Dup
    with app G show ?thesis
      by (cases s2, clarsimp simp add: sup_state_Cons2,
          auto dest: sup_state_length)
  next
    case Dup_x1
    with app G show ?thesis
      by (cases s2, clarsimp simp add: sup_state_Cons2,
          auto dest: sup_state_length)
  next
    case Dup_x2
    with app G show ?thesis
      by (cases s2, clarsimp simp add: sup_state_Cons2,
          auto dest: sup_state_length)
  next
    case Swap
    with app G show ?thesis
      by (cases s2, clarsimp simp add: sup_state_Cons2)
  next
    case IAdd
    with app G show ?thesis
      by (cases s2, auto simp add: sup_state_Cons2 PrimT_PrimT)
  next
    case Goto
    with app show ?thesis by simp
  next
    case Ifcmpeq
    with app G show ?thesis
      by (cases s2, auto simp add: sup_state_Cons2 PrimT_PrimT widen_RefT2)
  next
    case Invoke

    with app
    obtain apTs X ST LT mD' rT' b' where
      s1: "s1 = (rev apTs @ X # ST, LT)" and
      l:  "length apTs = length list" and
      c:  "is_class G cname" and
      C:  "G ⊢ X ⪯ Class cname" and
      w:  "∀x ∈ set (zip apTs list). x ∈ widen G" and
```

```
      m:  "method (G, cname) (mname, list) = Some (mD', rT', b')" and
      x:  "∀C ∈ set (match_any G pc et). is_class G C"
      by (simp del: not_None_eq, elim exE conjE) (rule that)

    obtain apTs' X' ST' LT' where
      s2: "s2 = (rev apTs' @ X' # ST', LT')" and
      l': "length apTs' = length list"
    proof -
      from l s1 G
      have "length list < length (fst s2)"
        by simp
      hence "∃a b c. (fst s2) = rev a @ b # c ∧ length a = length list"
        by (rule rev_append_cons [rule_format])
      thus ?thesis
        by -  (cases s2, elim exE conjE, simp, rule that)
    qed

    from l l'
    have "length (rev apTs') = length (rev apTs)" by simp

    from this s1 s2 G
    obtain
      G': "G ⊢ (apTs',LT') <=s (apTs,LT)" and
      X : "G ⊢  X' ≼ X" and "G ⊢ (ST',LT') <=s (ST,LT)"
      by (simp add: sup_state_rev_fst sup_state_append_fst sup_state_Cons1)

    with C
    have C': "G ⊢ X' ≼ Class cname"
      by - (rule widen_trans, auto)

    from G'
    have "G ⊢ map OK apTs' <=l map OK apTs"
      by (simp add: sup_state_conv)
    also
    from l w
    have "G ⊢ map OK apTs <=l map OK list"
      by (simp add: all_widen_is_sup_loc)
    finally
    have "G ⊢ map OK apTs' <=l map OK list" .

    with l'
    have w': "∀x ∈ set (zip apTs' list). x ∈ widen G"
      by (simp add: all_widen_is_sup_loc)

    from Invoke s2 l' w' C' m c x
    show ?thesis
      by (simp del: split_paired_Ex) blast
  next
    case Throw
    with app G show ?thesis
      by (cases s2, clarsimp simp add: sup_state_Cons2 widen_RefT2)
  qed
} note this [simp]
```

```
  assume "G ⊢ s <=' s'" "app i G m rT pc et s'"
  thus ?thesis by (cases s, cases s', auto)
qed

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 (cases s1, cases s2)
  fix a1 b1 a2 b2
  assume s: "s1 = (a1,b1)" "s2 = (a2,b2)"
  assume app2: "app i G m rT pc et (Some s2)"
  assume G: "G ⊢ s1 <=s s2"

  note [simp] = eff_def

  hence "G ⊢ (Some s1) <=' (Some s2)" by simp
  from this app2
  have app1: "app i G m rT pc et (Some s1)" by (rule app_mono)

  show ?thesis
  proof (cases (open) i)
    case Load

    with s app1
    obtain y where
       y:  "nat < length b1" "b1 ! nat = OK y" by clarsimp

    from Load s app2
    obtain y' where
       y': "nat < length b2" "b2 ! nat = OK y'" by clarsimp

    from G s
    have "G ⊢ b1 <=l b2" by (simp add: sup_state_conv)

    with y y'
    have "G ⊢ y ⊔ y'"
      by - (drule sup_loc_some, simp+)

    with Load G y y' s app1 app2
    show ?thesis by (clarsimp simp add: sup_state_conv)
  next
    case Store
    with G s app1 app2
    show ?thesis
      by (clarsimp simp add: sup_state_conv sup_loc_update)
  next
    case LitPush
    with G s app1 app2
    show ?thesis
      by (clarsimp simp add: sup_state_Cons1)
  next
    case New
```

```
    with G s app1 app2
    show ?thesis
      by (clarsimp simp add: sup_state_Cons1)
  next
    case Getfield
    with G s app1 app2
    show ?thesis
      by (clarsimp simp add: sup_state_Cons1)
  next
    case Putfield
    with G s app1 app2
    show ?thesis
      by (clarsimp simp add: sup_state_Cons1)
  next
    case Checkcast
    with G s app1 app2
    show ?thesis
      by (clarsimp simp add: sup_state_Cons1)
  next
    case Invoke

    with s app1
    obtain a X ST where
      s1: "s1 = (a @ X # ST, b1)" and
      l:  "length a = length list"
      by (simp, elim exE conjE, simp (no_asm_simp))

    from Invoke s app2
    obtain a' X' ST' where
      s2: "s2 = (a' @ X' # ST', b2)" and
      l': "length a' = length list"
      by (simp, elim exE conjE, simp (no_asm_simp))

    from l l'
    have lr: "length a = length a'" by simp

    from lr G s1 s2
    have "G ⊢ (ST, b1) <=s (ST', b2)"
      by (simp add: sup_state_append_fst sup_state_Cons1)

    moreover

    obtain b1' b2' where eff':
      "b1' = snd (eff' (i,G,s1))"
      "b2' = snd (eff' (i,G,s2))" by simp

    from Invoke G s eff' app1 app2
    obtain "b1 = b1'" "b2 = b2'" by simp

    ultimately

    have "G ⊢ (ST, b1') <=s (ST', b2')" by simp

    with Invoke G s app1 app2 eff' s1 s2 l l'
```

```
    show ?thesis
      by (clarsimp simp add: sup_state_conv)
  next
    case Return
    with G
    show ?thesis
      by simp
  next
    case Pop
    with G s app1 app2
    show ?thesis
      by (clarsimp simp add: sup_state_Cons1)
  next
    case Dup
    with G s app1 app2
    show ?thesis
      by (clarsimp simp add: sup_state_Cons1)
  next
    case Dup_x1
    with G s app1 app2
    show ?thesis
      by (clarsimp simp add: sup_state_Cons1)
  next
    case Dup_x2
    with G s app1 app2
    show ?thesis
      by (clarsimp simp add: sup_state_Cons1)
  next
    case Swap
    with G s app1 app2
    show ?thesis
      by (clarsimp simp add: sup_state_Cons1)
  next
    case IAdd
    with G s app1 app2
    show ?thesis
      by (clarsimp simp add: sup_state_Cons1)
  next
    case Goto
    with G s app1 app2
    show ?thesis by simp
  next
    case Ifcmpeq
    with G s app1 app2
    show ?thesis
      by (clarsimp simp add: sup_state_Cons1)
  next
    case Throw
    with G
    show ?thesis
      by simp
  qed
qed
```

lemmas *[iff del] = not_Err_eq*

end

4.14 The Bytecode Verifier

theory *BVSpec = Effect:*

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

constdefs

— The program counter will always be inside the method:

```
check_bounded :: "instr list ⇒ exception_table ⇒ bool"
"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)"
```

— The method type only contains declared classes:

```
check_types :: "jvm_prog ⇒ nat ⇒ nat ⇒ state list ⇒ bool"
"check_types G mxs mxr phi ≡ set phi ⊆ states G mxs mxr"
```

— 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"
"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')"
```

— The type at *pc=0* conforms to the method calling convention:

```
wt_start :: "[jvm_prog,cname,ty list,nat,method_type] ⇒ bool"
"wt_start G C pTs mxl phi ==
G ⊢ Some ([],(OK (Class C))#((map OK pTs))@(replicate mxl Err)) <=' phi!0"
```

— 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"
"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)"
```

— A program is welltyped if it is wellformed and all methods are welltyped

```
wt_jvm_prog :: "[jvm_prog,prog_type] ⇒ bool"
"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"
```

apply *(unfold eff_def)*
apply *simp*
apply *(unfold check_bounded_def)*
apply *clarify*
apply *(erule disjE)*
apply *blast*
apply *(erule allE, erule impE, assumption)*
apply *(unfold xcpt_eff_def)*
apply *clarsimp*
apply *(drule xcpt_names_in_et)*
apply *clarify*
apply *(drule bspec, assumption)*
apply *simp*
done

lemma *wt_jvm_progD:*

```
  "wt_jvm_prog G phi ⟹ (∃wt. wf_prog wt G)"
```

by *(unfold wt_jvm_prog_def, blast)*

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"
```

by *(unfold wt_jvm_prog_def, drule method_wf_mdecl,*
simp, simp, simp add: wf_mdecl_def wt_method_def)

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'))"
```

apply *(simp add: wt_instr_def)*
apply *(unfold wt_jvm_prog_def)*
apply *(drule method_wf_mdecl)*
apply *(simp, simp, simp add: wf_mdecl_def wt_method_def)*
apply *(auto dest: check_boundedD)*
done

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)"
```
by *(unfold wt_jvm_prog_def, drule method_wf_mdecl,*
simp, simp, simp add: wf_mdecl_def wt_method_def)

end

4.15 The Typing Framework for the JVM

theory *Typing_Framework_JVM = Typing_Framework_err + JVMType + EffectMono + BVSpec:*

constdefs
exec :: "jvm_prog ⇒ nat ⇒ ty ⇒ exception_table ⇒ instr list ⇒ state step_type"
"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)"

constdefs
opt_states :: "'c prog ⇒ nat ⇒ nat ⇒ (ty list × ty err list) option set"
"opt_states G maxs maxr ≡ opt (⋃{list n (types G) |n. n ≤ maxs} × list maxr (err (types G)))"

4.15.1 Executability of *check_bounded*

consts
list_all'_rec :: "('a ⇒ nat ⇒ bool) ⇒ nat ⇒ 'a list ⇒ bool"
primrec
"list_all'_rec P n []   = True"
"list_all'_rec P n (x#xs) = (P x n ∧ list_all'_rec P (Suc n) xs)"

constdefs
list_all' :: "('a ⇒ nat ⇒ bool) ⇒ 'a list ⇒ bool"
"list_all' P xs ≡ list_all'_rec P 0 xs"

lemma *list_all'_rec:*
"⋀n. list_all'_rec P n xs = (∀p < size xs. P (xs!p) (p+n))"
apply *(induct xs)*
apply *auto*
apply *(case_tac p)*
apply *auto*
done

lemma *list_all' [iff]:*
"list_all' P xs = (∀n < size xs. P (xs!n) n)"
by *(unfold list_all'_def) (simp add: list_all'_rec)*

lemma *list_all_ball:*
"list_all P xs = (∀x ∈ set xs. P x)"
by *(induct xs) auto*

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)"
by *(simp add: list_all_ball check_bounded_def)*

4.15.2 Connecting JVM and Framework

lemma *check_bounded_is_bounded:*
"check_bounded ins et ⟹ bounded (λpc. eff (ins!pc) G pc et) (length ins)"
by *(unfold bounded_def) (blast dest: check_boundedD)*

```
lemma special_ex_swap_lemma [iff]:
  "(? X. (? n. X = A n & P n) & Q X) = (? n. Q(A n) & P n)"
  by blast

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)"
  apply (unfold exec_def JVM_states_unfold)
  apply (rule pres_type_lift)
  apply clarify
  apply (case_tac s)
   apply simp
   apply (drule effNone)
   apply simp
  apply (simp add: eff_def xcpt_eff_def norm_eff_def)
  apply (case_tac "bs!p")

  apply (clarsimp simp add: not_Err_eq)
  apply (drule listE_nth_in, assumption)
  apply fastsimp

  apply (fastsimp simp add: not_None_eq)

  apply (fastsimp simp add: not_None_eq typeof_empty_is_type)

  apply clarsimp
  apply (erule disjE)
   apply fastsimp
  apply clarsimp
  apply (rule_tac x="1" in exI)
  apply fastsimp

  apply clarsimp
  apply (erule disjE)
   apply (fastsimp dest: field_fields fields_is_type)
  apply (simp add: match_some_entry split: split_if_asm)
  apply (rule_tac x=1 in exI)
  apply fastsimp

  apply clarsimp
  apply (erule disjE)
   apply fastsimp
  apply (simp add: match_some_entry split: split_if_asm)
  apply (rule_tac x=1 in exI)
  apply fastsimp

  apply clarsimp
  apply (erule disjE)
   apply fastsimp
  apply clarsimp
  apply (rule_tac x=1 in exI)
  apply fastsimp
```

```
  defer

  apply fastsimp
  apply fastsimp

  apply clarsimp
  apply (rule_tac x="n'+2" in exI)
  apply simp

  apply clarsimp
  apply (rule_tac x="Suc (Suc (Suc (length ST)))" in exI)
  apply simp

  apply clarsimp
  apply (rule_tac x="Suc (Suc (Suc (Suc (length ST))))" in exI)
  apply simp

  apply fastsimp
  apply fastsimp
  apply fastsimp
  apply fastsimp

  apply clarsimp
  apply (erule disjE)
   apply fastsimp
  apply clarsimp
  apply (rule_tac x=1 in exI)
  apply fastsimp

  apply (erule disjE)
   apply (clarsimp simp add: Un_subset_iff)
   apply (drule method_wf_mdecl, assumption+)
   apply (clarsimp simp add: wf_mdecl_def wf_mhead_def)
   apply fastsimp
  apply clarsimp
  apply (rule_tac x=1 in exI)
  apply fastsimp
  done

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))))"
  by (simp add: sup_state_opt_def sup_state_def sup_loc_def sup_ty_opt_def)


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)"
  by (unfold app_mono_def lesub_def) (blast intro: EffectMono.app_mono)
```

lemma *list_appendI:*
"⟦a ∈ list x A; b ∈ list y A⟧ ⟹ a @ b ∈ list (x+y) A"
apply *(unfold list_def)*
apply *(simp (no_asm))*
apply *blast*
done

lemma *list_map [simp]:*
"(map f xs ∈ list (length xs) A) = (f ` set xs ⊆ A)"
apply *(unfold list_def)*
apply *simp*
done

lemma *[iff]:*
"(OK ` A ⊆ err B) = (A ⊆ B)"
apply *(unfold err_def)*
apply *blast*
done

lemma *[intro]:*
"x ∈ A ⟹ replicate n x ∈ list n A"
by *(induct n, auto)*

lemma *lesubstep_type_simple:*
"a <=[Product.le (op =) r] b ⟹ a <=|r| b"
apply *(unfold lesubstep_type_def)*
apply *clarify*
apply *(simp add: set_conv_nth)*
apply *clarify*
apply *(drule le_listD, assumption)*
apply *(clarsimp simp add: lesub_def Product.le_def)*
apply *(rule exI)*
apply *(rule conjI)*
apply *(rule exI)*
apply *(rule conjI)*
apply *(rule sym)*
apply *assumption*
apply *assumption*
apply *assumption*
done

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"
apply *(unfold eff_def)*
apply *(rule lesubstep_type_simple)*
apply *(rule le_list_appendI)*
apply *(simp add: norm_eff_def)*
apply *(rule le_listI)*
apply *simp*
apply *simp*
apply *(simp add: lesub_def)*
apply *(case_tac s)*

```
    apply simp
   apply (simp del: split_paired_All split_paired_Ex)
   apply (elim exE conjE)
   apply simp
   apply (drule eff'_mono, assumption)
   apply assumption
  apply (simp add: xcpt_eff_def)
  apply (rule le_listI)
    apply simp
  apply simp
  apply (simp add: lesub_def)
  apply (case_tac s)
   apply simp
  apply simp
  apply (case_tac t)
   apply simp
  apply (clarsimp simp add: sup_state_conv)
  done

lemma order_sup_state_opt:
  "wf_prog wf_mb G ⟹ order (sup_state_opt G)"
  by (unfold sup_state_opt_unfold) (blast dest: acyclic_subcls1 order_widen)

theorem exec_mono:
  "wf_prog wf_mb 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)"

  apply (unfold exec_def JVM_le_unfold JVM_states_unfold)
  apply (rule mono_lift)
     apply (fold sup_state_opt_unfold opt_states_def)
     apply (erule order_sup_state_opt)
    apply (rule app_mono)
   apply assumption
  apply clarify
  apply (rule eff_mono)
  apply assumption+
  done

theorem semilat_JVM_slI:
  "wf_prog wf_mb G ⟹ semilat (JVMType.sl G maxs maxr)"
  apply (unfold JVMType.sl_def stk_esl_def reg_sl_def)
  apply (rule semilat_opt)
  apply (rule err_semilat_Product_esl)
  apply (rule err_semilat_upto_esl)
  apply (rule err_semilat_JType_esl, assumption+)
  apply (rule err_semilat_eslI)
  apply (rule Listn_sl)
  apply (rule err_semilat_JType_esl, assumption+)
  done

lemma sl_triple_conv:
  "JVMType.sl G maxs maxr ==
  (states G maxs maxr, JVMType.le G maxs maxr, JVMType.sup G maxs maxr)"
  by (simp (no_asm) add: states_def JVMType.le_def JVMType.sup_def)
```

```
lemma map_id [rule_format]:
  "(∀n < length xs. f (g (xs!n)) = xs!n) ⟶ map f (map g xs) = xs"
  by (induct xs, auto)


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
  assume "wf_prog wf_mb G"
         "(C,S,fs,mdecls) ∈ set G"
         "((mn,pTs),rT,code) ∈ set mdecls"
  hence "wf_mdecl wf_mb G C ((mn,pTs),rT,code)"
    by (unfold wf_prog_def wf_cdecl_def) auto
  hence "∀t ∈ set pTs. is_type G t"
    by (unfold wf_mdecl_def wf_mhead_def) auto
  moreover
  fix t assume "t ∈ set pTs"
  ultimately
  have "is_type G t" by blast
  thus "t ∈ types G" ..
qed


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 -
  from wf show ?thesis
    apply (unfold wf_prog_def wf_cdecl_def)
    apply clarsimp
    apply (drule bspec, assumption)
    apply (unfold wf_mdecl_def)
    apply clarsimp
    apply (drule bspec, assumption)
    apply clarsimp
    apply (frule methd [OF wf], assumption+)
    apply (frule is_type_pTs [OF wf], assumption+)
    apply clarify
```

```
    apply (drule rule [OF wf], assumption+)
    apply (rule refl)
    apply assumption+
    done
qed

end
```

4.16 Kildall for the JVM

theory *JVM = Kildall + Typing_Framework_JVM:*

constdefs
```
  kiljvm :: "jvm_prog ⇒ nat ⇒ nat ⇒ ty ⇒ exception_table ⇒
             instr list ⇒ state list ⇒ state list"
  "kiljvm G maxs maxr rT et bs ==
  kildall (JVMType.le G maxs maxr) (JVMType.sup G maxs maxr) (exec G maxs rT et bs)"

  wt_kil :: "jvm_prog ⇒ cname ⇒ ty list ⇒ ty ⇒ nat ⇒ nat ⇒
             exception_table ⇒ instr list ⇒ bool"
  "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)"

  wt_jvm_prog_kildall :: "jvm_prog ⇒ bool"
  "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)"
  apply (unfold kiljvm_def sl_triple_conv)
  apply (rule is_bcv_kildall)
       apply (simp (no_asm) add: sl_triple_conv [symmetric])
       apply (force intro!: semilat_JVM_slI dest: wf_acyclic simp add: symmetric sl_triple_conv)
      apply (simp (no_asm) add: JVM_le_unfold)
      apply (blast intro!: order_widen wf_converse_subcls1_impl_acc_subtype
                   dest: wf_subcls1 wf_acyclic)
     apply (simp add: JVM_le_unfold)
    apply (erule exec_pres_type)
   apply assumption
  apply (erule exec_mono, assumption)
  done

lemma subset_replicate: "set (replicate n x) ⊆ {x}"
  by (induct n) auto

lemma in_set_replicate:
  "x ∈ set (replicate n y) ⟹ x = y"
proof -
  assume "x ∈ set (replicate n y)"
  also have "set (replicate n y) ⊆ {y}" by (rule subset_replicate)
  finally have "x ∈ {y}" .
  thus ?thesis by simp
qed
```

```
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 -
  let ?start = "OK (Some ([],(OK (Class C))#((map OK pTs))@(replicate mxl Err)))
                #(replicate (size bs - 1) (OK None))"

  from wtk obtain maxr r where
    bounded: "check_bounded bs et" and
    result:  "r = kiljvm G maxs maxr rT et bs ?start" and
    success: "∀n < size bs. r!n ≠ Err" and
    instrs:  "0 < size bs" and
    maxr:    "maxr = Suc (length pTs + mxl)"
    by (unfold wt_kil_def) simp

  from bounded have "bounded (exec G maxs rT et bs) (size bs)"
    by (unfold exec_def) (intro bounded_lift check_bounded_is_bounded)
  with wf have bcv:
    "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)"
    by (rule is_bcv_kiljvm)

  from C pTs instrs maxr
  have "?start ∈ list (length bs) (states G maxs maxr)"
    apply (unfold JVM_states_unfold)
    apply (rule listI)
    apply (auto intro: list_appendI dest!: in_set_replicate)
    apply force
    done

  with bcv success result have
    "∃ts∈list (length bs) (states G maxs maxr).
         ?start <=[JVMType.le G maxs maxr] ts ∧
         wt_step (JVMType.le G maxs maxr) Err (exec G maxs rT et bs) ts"
    by (unfold is_bcv_def) auto
  then obtain phi' where
    phi': "phi' ∈ list (length bs) (states G maxs maxr)" and
    s: "?start <=[JVMType.le G maxs maxr] phi'" and
    w: "wt_step (JVMType.le G maxs maxr) Err (exec G maxs rT et bs) phi'"
    by blast
  hence wt_err_step:
    "wt_err_step (sup_state_opt G) (exec G maxs rT et bs) phi'"
    by (simp add: wt_err_step_def exec_def JVM_le_Err_conv)

  from s have le: "JVMType.le G maxs maxr (?start ! 0) (phi'!0)"
    by (drule_tac p=0 in le_listD) (simp add: lesub_def)+

  from phi' have l: "size phi' = size bs" by simp
```

```
  with instrs w have "phi' ! 0 ≠ Err" by (unfold wt_step_def) simp
  with instrs l have phi0: "OK (map ok_val phi' ! 0) = phi' ! 0"
    by (clarsimp simp add: not_Err_eq)

  from phi' have "check_types G maxs maxr phi'" by(simp add: check_types_def)
  also from w have "phi' = map OK (map ok_val phi')"
    apply (clarsimp simp add: wt_step_def not_Err_eq)
    apply (rule map_id [symmetric])
    apply (erule allE, erule impE, assumption)
    apply clarsimp
    done
  finally
  have check_types:
    "check_types G maxs maxr (map OK (map ok_val phi'))" .

  from l bounded
  have "bounded (λpc. eff (bs!pc) G pc et) (length phi')"
    by (simp add: exec_def check_bounded_is_bounded)
  hence bounded': "bounded (exec G maxs rT et bs) (length bs)"
    by (auto intro: bounded_lift simp add: exec_def l)
  with wt_err_step
  have "wt_app_eff (sup_state_opt G) (λpc. app (bs!pc) G maxs rT pc et)
                   (λpc. eff (bs!pc) G pc et) (map ok_val phi')"
    by (auto intro: wt_err_imp_wt_app_eff simp add: l exec_def)
  with instrs l le bounded bounded' check_types maxr
  have "wt_method G C pTs rT maxs mxl bs et (map ok_val phi')"
    apply (unfold wt_method_def wt_app_eff_def)
    apply simp
    apply (rule conjI)
     apply (unfold wt_start_def)
     apply (rule JVM_le_convert [THEN iffD1])
     apply (simp (no_asm) add: phi0)
    apply clarify
    apply (erule allE, erule impE, assumption)
    apply (elim conjE)
    apply (clarsimp simp add: lesub_def wt_instr_def)
    apply (simp add: exec_def)
    apply (drule bounded_err_stepD, assumption+)
    apply blast
    done

  thus ?thesis by blast
qed


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 -
```

```
let ?mxr = "1+size pTs+mxl"

from wtm obtain
  instrs:   "0 < length bs" and
  len:      "length phi = length bs" and
  bounded:  "check_bounded bs et" and
  ck_types: "check_types G maxs ?mxr (map OK phi)" and
  wt_start: "wt_start G C pTs mxl phi" and
  wt_ins:   "∀pc. pc < length bs ⟶
                wt_instr (bs ! pc) G rT phi maxs (length bs) et pc"
  by (unfold wt_method_def) simp

from ck_types len
have istype_phi:
  "map OK phi ∈ list (length bs) (states G maxs (1+size pTs+mxl))"
  by (auto simp add: check_types_def intro!: listI)

let ?eff  = "λpc. eff (bs!pc) G pc et"
let ?app  = "λpc. app (bs!pc) G maxs rT pc et"

from bounded
have bounded_exec: "bounded (exec G maxs rT et bs) (size bs)"
  by (unfold exec_def) (intro bounded_lift check_bounded_is_bounded)

from wt_ins
have "wt_app_eff (sup_state_opt G) ?app ?eff phi"
  apply (unfold wt_app_eff_def wt_instr_def lesub_def)
  apply (simp (no_asm) only: len)
  apply blast
  done
with bounded_exec
have "wt_err_step (sup_state_opt G) (err_step (size phi) ?app ?eff) (map OK phi)"
  by - (erule wt_app_eff_imp_wt_err,simp add: exec_def len)
hence wt_err:
  "wt_err_step (sup_state_opt G) (exec G maxs rT et bs) (map OK phi)"
  by (unfold exec_def) (simp add: len)

from wf bounded_exec
have is_bcv:
  "is_bcv (JVMType.le G maxs ?mxr) Err (exec G maxs rT et bs)
          (size bs) (states G maxs ?mxr) (kiljvm G maxs ?mxr rT et bs)"
  by (rule is_bcv_kiljvm)

let ?start = "OK (Some ([],(OK (Class C))#((map OK pTs))@(replicate mxl Err)))
              #(replicate (size bs - 1) (OK None))"

from C pTs instrs
have start: "?start ∈ list (length bs) (states G maxs ?mxr)"
  apply (unfold JVM_states_unfold)
  apply (rule listI)
  apply (auto intro!: list_appendI dest!: in_set_replicate)
  apply force
  done
```

```
  let ?phi = "map OK phi"
  have less_phi: "?start <=[JVMType.le G maxs ?mxr] ?phi"
  proof -
    from len instrs
    have "length ?start = length (map OK phi)" by simp
    moreover
    { fix n
      from wt_start
      have "G ⊢ ok_val (?start!0) <=' phi!0"
        by (simp add: wt_start_def)
      moreover
      from instrs len
      have "0 < length phi" by simp
      ultimately
      have "JVMType.le G maxs ?mxr (?start!0) (?phi!0)"
        by (simp add: JVM_le_Err_conv Err.le_def lesub_def)
      moreover
      { fix n'
        have "JVMType.le G maxs ?mxr (OK None) (?phi!n)"
          by (auto simp add: JVM_le_Err_conv Err.le_def lesub_def
            split: err.splits)
        hence "⟦ n = Suc n'; n < length ?start ⟧
          ⟹ JVMType.le G maxs ?mxr (?start!n) (?phi!n)"
          by simp
      }
      ultimately
      have "n < length ?start ⟹ (?start!n) <=_(JVMType.le G maxs ?mxr) (?phi!n)"
        by (unfold lesub_def) (cases n, blast+)
    }
    ultimately show ?thesis by (rule le_listI)
  qed

  from wt_err
  have "wt_step (JVMType.le G maxs ?mxr) Err (exec G maxs rT et bs) ?phi"
    by (simp add: wt_err_step_def JVM_le_Err_conv)
  with start istype_phi less_phi is_bcv
  have "∀p. p < length bs ⟶ kiljvm G maxs ?mxr rT et bs ?start ! p ≠ Err"
    by (unfold is_bcv_def) auto
  with bounded instrs
  show "wt_kil G C pTs rT maxs mxl et bs" by (unfold wt_kil_def) simp
qed


theorem jvm_kildall_sound_complete:
  "wt_jvm_prog_kildall G = (∃Phi. wt_jvm_prog G Phi)"
proof
  let ?Phi = "λC sig. let (C,rT,(maxs,maxl,ins,et)) = the (method (G,C) sig) in
              SOME phi. wt_method G C (snd sig) rT maxs maxl ins et phi"

  assume "wt_jvm_prog_kildall G"
  hence "wt_jvm_prog G ?Phi"
    apply (unfold wt_jvm_prog_def wt_jvm_prog_kildall_def)
    apply (erule jvm_prog_lift)
    apply (auto dest!: wt_kil_correct intro: someI)
```

```
    done
  thus "∃Phi. wt_jvm_prog G Phi" by fast
next
  assume "∃Phi. wt_jvm_prog G Phi"
  thus "wt_jvm_prog_kildall G"
    apply (clarify)
    apply (unfold wt_jvm_prog_def wt_jvm_prog_kildall_def)
    apply (erule jvm_prog_lift)
    apply (auto intro: wt_kil_complete)
    done
qed

end
```

4.17 The Lightweight Bytecode Verifier

theory *LBVSpec = SemilatAlg + Opt:*

types
```
  's certificate = "'s list"
```

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

constdefs
```
wtl_inst :: "'s certificate ⇒ 's binop ⇒ 's ord ⇒ 's ⇒
             's step_type ⇒ nat ⇒ 's ⇒ 's"
"wtl_inst cert f r T step pc s ≡ merge cert f r T pc (step pc s) (cert!(pc+1))"

wtl_cert :: "'s certificate ⇒ 's binop ⇒ 's ord ⇒ 's ⇒ 's ⇒
             's step_type ⇒ nat ⇒ 's ⇒ 's"
"wtl_cert cert f r T B step pc s ≡
  if cert!pc = B then
    wtl_inst cert f r T step pc s
  else
    if s <=_r (cert!pc) then wtl_inst cert f r T step pc (cert!pc) else T"
```

consts
```
wtl_inst_list :: "'a list ⇒ 's certificate ⇒ 's binop ⇒ 's ord ⇒ 's ⇒ 's ⇒
                  's step_type ⇒ nat ⇒ 's ⇒ 's"
```
primrec
```
"wtl_inst_list []     cert f r T B step pc s = s"
"wtl_inst_list (i#is) cert f r T B step pc s =
    (let s' = wtl_cert cert f r T B step pc s in
      if s' = T ∨ s = T then T else wtl_inst_list is cert f r T B step (pc+1) s')"
```

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

constdefs
```
  bottom :: "'a ord ⇒ 'a ⇒ bool"
  "bottom r B ≡ ∀x. B <=_r x"
```

locale (**open**) *lbv = semilat +*
  fixes *T :: "'a" ("⊤")*
  fixes *B :: "'a" ("⊥")*
  fixes *step :: "'a step_type"*
  assumes *top: "top r ⊤"*
  assumes *T_A: "⊤ ∈ A"*

assumes *bot: "bottom r ⊥"*
assumes *B_A: "⊥ ∈ 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))"
by *(simp add: wti_def mrg_def wtl_inst_def)*

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 ⊤"
by *(unfold wtc_def wti_def wtl_cert_def)*

lemma *cert_okD1 [intro?]:*
"cert_ok c n T B A ⟹ pc < n ⟹ c!pc ∈ A"
by *(unfold cert_ok_def) fast*

lemma *cert_okD2 [intro?]:*
"cert_ok c n T B A ⟹ c!n = B"
by *(simp add: cert_ok_def)*

lemma *cert_okD3 [intro?]:*
"cert_ok c n T B A ⟹ B ∈ A ⟹ pc < n ⟹ c!Suc pc ∈ A"
by *(drule Suc_leI) (auto simp add: le_eq_less_or_eq dest: cert_okD1 cert_okD2)*

lemma *cert_okD4 [intro?]:*
"cert_ok c n T B A ⟹ pc < n ⟹ c!pc ≠ T"
by *(simp add: cert_ok_def)*

declare *Let_def [simp]*

4.17.1 more semilattice lemmas

lemma (**in** *lbv*) *sup_top [simp, elim]:*
assumes *x: "x ∈ A"*
shows *"x +_f ⊤ = ⊤"*
proof -
from *top* **have** *"x +_f ⊤ <=_r ⊤"* **..**
moreover from *x* **have** *"⊤ <=_r x +_f ⊤"* **..**
ultimately show *?thesis* **..**
qed

lemma (**in** *lbv*) *plusplussup_top [simp, elim]:*
  "set xs ⊆ A ⟹ xs ++_f ⊤ = ⊤"
  by (*induct xs*) *auto*

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 -
  from *S* **have** *"∀(x,y) ∈ set S. y ∈ A"* **by** *auto*
  with *semilat y ab* **show** *?thesis* **by** - (*rule ub1'*)
qed

lemma (**in** *lbv*) *bottom_le [simp, intro]:*
  "⊥ <=_r x"
  by (*insert bot*) (*simp add: bottom_def*)

lemma (**in** *lbv*) *le_bottom [simp]:*
  "x <=_r ⊥ = (x = ⊥)"
  by (*blast intro: antisym_r*)

4.17.2 merge

lemma (**in** *lbv*) *merge_Nil [simp]:*
  "merge c pc [] x = x" **by** (*simp add: mrg_def*)

lemma (**in** *lbv*) *merge_Cons [simp]:*
  "merge c pc (l#ls) x = merge c pc ls (if fst l=pc+1 then snd l +_f x
                                        else if snd l <=_r (c!fst l) then x
                                        else ⊤)"
  by (*simp add: mrg_def split_beta*)

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

lemma (**in** *lbv*) *merge_not_top:*
  "⋀x. snd`set ss ⊆ A ⟹ merge c pc ss x ≠ ⊤ ⟹
  ∀(pc',s') ∈ set ss. (pc' ≠ pc+1 ⟶ s' <=_r (c!pc'))"
  (**is** *"⋀x. ?set ss ⟹ ?merge ss x ⟹ ?P ss"*)
proof (*induct ss*)
  show *"?P []"* **by** *simp*
next
  fix *x ls l*
  assume *"?set (l#ls)"* **then obtain** *set: "snd`set ls ⊆ A"* **by** *simp*
  assume *merge: "?merge (l#ls) x"*
  moreover
  obtain *pc' s'* **where** *[simp]: "l = (pc',s')"* **by** (*cases l*)
  ultimately
  obtain *x'* **where** *"?merge ls x'"* **by** *simp*
  assume *"⋀x. ?set ls ⟹ ?merge ls x ⟹ ?P ls"* **hence** *"?P ls"* .

```
  moreover
  from merge set
  have "pc' ≠ pc+1 ⟶ s' <=_r (c!pc')" by (simp split: split_if_asm)
  ultimately
  show "?P (l#ls)" by simp
qed


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 (induct ss)
  fix x show "?P [] x" by simp
next
  fix x assume x: "x ∈ A"
  fix l::"nat × 'a" and ls
  assume "snd`set (l#ls) ⊆ A"
  then obtain l: "snd l ∈ A" and ls: "snd`set ls ⊆ A" by auto
  assume "⋀x. x ∈ A ⟹ snd`set ls ⊆ A ⟹ ?P ls x"
  hence IH: "⋀x. x ∈ A ⟹ ?P ls x" .
  obtain pc' s' where [simp]: "l = (pc',s')" by (cases l)
  hence "?merge (l#ls) x = ?merge ls
    (if pc'=pc+1 then s' +_f x else if s' <=_r c!pc' then x else ⊤)"
    (is "?merge (l#ls) x = ?merge ls ?if'")
    by simp
  also have "... = ?if ls ?if'"
  proof -
    from l have "s' ∈ A" by simp
    with x have "s' +_f x ∈ A" by simp
    with x have "?if' ∈ A" by auto
    hence "?P ls ?if'" by (rule IH) thus ?thesis by simp
  qed
  also have "... = ?if (l#ls) x"
    proof (cases "∀(pc', s')∈set (l#ls). pc'≠pc+1 ⟶ s' <=_r c!pc'")
      case True
      hence "∀(pc', s')∈set ls. pc'≠pc+1 ⟶ s' <=_r c!pc'" by auto
      moreover
      from True have
        "map snd [(p',t')∈ls . p'=pc+1] ++_f ?if' =
        (map snd [(p',t')∈l#ls . p'=pc+1] ++_f x)"
        by simp
      ultimately
      show ?thesis using True by simp
    next
      case False
      moreover
      from ls have "set (map snd [(p', t')∈ls . p' = Suc pc]) ⊆ A" by auto
      ultimately show ?thesis by auto
    qed
```

finally show *"?P (l#ls) x"* .
qed

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 -
from *ss m* **have** *"∀(pc',s') ∈ set ss. (pc' ≠ pc+1 ⟶ s' <=_r c!pc')"*
by *(rule merge_not_top)*
with *x ss m* **show** *?thesis* **by** - *(drule merge_def, auto split: split_if_asm)*
qed

4.17.3 wtl-inst-list

lemmas *[iff] = not_Err_eq*

lemma (**in** *lbv*) *wtl_Nil [simp]: "wtl [] c pc s = s"*
by *(simp add: wtl_def)*

lemma (**in** *lbv*) *wtl_Cons [simp]:*
"wtl (i#is) c pc s =
(let s' = wtc c pc s in if s' = ⊤ ∨ s = ⊤ then ⊤ else wtl is c (pc+1) s')"
by *(simp add: wtl_def wtc_def)*

lemma (**in** *lbv*) *wtl_Cons_not_top:*
"wtl (i#is) c pc s ≠ ⊤ =
(wtc c pc s ≠ ⊤ ∧ s ≠ T ∧ wtl is c (pc+1) (wtc c pc s) ≠ ⊤)"
by *(auto simp del: split_paired_Ex)*

lemma (**in** *lbv*) *wtl_top [simp]:* *"wtl ls c pc ⊤ = ⊤"*
by *(cases ls) auto*

lemma (**in** *lbv*) *wtl_not_top:*
"wtl ls c pc s ≠ ⊤ ⟹ s ≠ ⊤"
by *(cases "s=⊤") auto*

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)"
by *(induct a) auto*

lemma (**in** *lbv*) *wtl_take:*
"wtl is c pc s ≠ ⊤ ⟹ wtl (take pc' is) c pc s ≠ ⊤"
*(***is** *"?wtl is ≠ _ ⟹ _")*
proof -
assume *"?wtl is ≠ ⊤"*
hence *"?wtl (take pc' is @ drop pc' is) ≠ ⊤"* **by** *simp*
thus *?thesis* **by** *(auto dest!: wtl_not_top simp del: append_take_drop_id)*
qed

lemma *take_Suc:*
"∀n. n < length l ⟶ take (Suc n) l = (take n l)@[l!n]" (**is** *"?P l"*)
proof *(induct l)*
show *"?P []"* **by** *simp*

```
next
  fix x xs assume IH: "?P xs"
  show "?P (x#xs)"
  proof (intro strip)
    fix n assume "n < length (x#xs)"
    with IH show "take (Suc n) (x # xs) = take n (x # xs) @ [(x # xs) ! n]"
      by (cases n, auto)
  qed
qed

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 -
  from suc have "take (pc+1) is=(take pc is)@[is!pc]" by (simp add: take_Suc)
  with suc wtl show ?thesis by (simp add: min_def)
qed

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 -
  from pc have "0 < length (drop pc is)" by simp
  then  obtain i r where Cons: "drop pc is = i#r"
    by (auto simp add: neq_Nil_conv simp del: length_drop)
  hence "i#r = drop pc is" ..
  with all have take: "?wtl (take pc is@i#r) ≠ ⊤" by simp
  from pc have "is!pc = drop pc is ! 0" by simp
  with Cons have "is!pc = i" by simp
  with take pc show ?thesis by (auto simp add: min_def split: split_if_asm)
qed
```

4.17.4 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 -
  from s0 have "set (map snd [(p', t')∈ss . p'=pc+1]) ⊆ A" by auto
  with x  have "(map snd [(p', t')∈ss . p'=pc+1] ++_f x) ∈ A"
    by (auto intro!: plusplus_closed)
  with s0 x show ?thesis by (simp add: merge_def T_A)
qed


lemma pres_typeD2:
  "pres_type step n A ⟹ s ∈ A ⟹ p < n ⟹ snd`set (step p s) ⊆ A"
  by auto (drule pres_typeD)


lemma (in lbv) wti_pres [intro?]:
  assumes pres: "pres_type step n A"
```

```
  assumes cert: "c!(pc+1) ∈ A"
  assumes s_pc: "s ∈ A" "pc < n"
  shows "wti c pc s ∈ A"
proof -
  from pres s_pc have "snd`set (step pc s) ⊆ A" by (rule pres_typeD2)
  with cert show ?thesis by (simp add: wti merge_pres)
qed


lemma (in lbv) wtc_pres:
  assumes "pres_type step n A"
  assumes "c!pc ∈ A" and "c!(pc+1) ∈ A"
  assumes "s ∈ A" and "pc < n"
  shows "wtc c pc s ∈ A"
proof -
  have "wti c pc s ∈ A" ..
  moreover have "wti c pc (c!pc) ∈ A" ..
  ultimately show ?thesis using T_A by (simp add: wtc)
qed


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 (induct pc)
  from s show "?wtl 0 ∈ A" by simp
next
  fix n assume "Suc n < length is"
  then obtain n: "n < length is" by simp
  assume "n < length is ⟹ ?wtl n ∈ A"
  hence "?wtl n ∈ A" .
  moreover
  from cert have "c!n ∈ A" by (rule cert_okD1)
  moreover
  have n1: "n+1 < length is" by simp
  with cert have "c!(n+1) ∈ A" by (rule cert_okD1)
  ultimately
  have "wtc c n (?wtl n) ∈ A" by - (rule wtc_pres)
  also
  from all n have "?wtl n ≠ ⊤" by - (rule wtl_take)
  with n1 have "wtc c n (?wtl n) = ?wtl (n+1)" by (rule wtl_Suc [symmetric])
  finally  show "?wtl (Suc n) ∈ A" by simp
qed


end
```

4.18 Correctness of the LBV

theory *LBVCorrect = LBVSpec + Typing_Framework:*

locale *(open) lbvs = lbv +*
  fixes *s0 :: 'a ("s_0")*
  fixes *c :: "'a list"*
  fixes *ins :: "'b list"*
  fixes *phi :: "'a list" ("φ")*
  defines *phi_def:*
  "$\varphi \equiv$ map (λpc. if c!pc = $\bot$ 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) $\top$ $\bot$ A"*
  assumes *pres: "pres_type step (length ins) A"*

lemma *(in lbvs) phi_None [intro?]:*
  "$[\![$ pc < length ins; c!pc = $\bot$ $]\!]$ $\Longrightarrow$ φ ! pc = wtl (take pc ins) c 0 s0"
  by *(simp add: phi_def)*

lemma *(in lbvs) phi_Some [intro?]:*
  "$[\![$ pc < length ins; c!pc $\neq$ $\bot$ $]\!]$ $\Longrightarrow$ φ ! pc = c ! pc"
  by *(simp add: phi_def)*

lemma *(in lbvs) phi_len [simp]:*
  "length φ = length ins"
  by *(simp add: phi_def)*

lemma *(in lbvs) wtl_suc_pc:*
  assumes *all: "wtl ins c 0 s_0 $\neq$ $\top$"*
  assumes *pc: "pc+1 < length ins"*
  shows *"wtl (take (pc+1) ins) c 0 s0 $\leq_r$ φ!(pc+1)"*
proof *-*
  from *all pc*
  have *"wtc c (pc+1) (wtl (take (pc+1) ins) c 0 s0) $\neq$ T"* **by** *(rule wtl_all)*
  with *pc* **show** *?thesis* **by** *(simp add: phi_def wtc split: split_if_asm)*
qed

lemma *(in lbvs) wtl_stable:*
  assumes *wtl: "wtl ins c 0 s0 $\neq$ $\top$"*
  assumes *s0: "s0 $\in$ A"*
  assumes *pc: "pc < length ins"*
  shows *"stable r step φ pc"*
proof *(unfold stable_def, clarify)*
  fix *pc' s'* **assume** *step: "(pc',s') $\in$ set (step pc (φ ! pc))"*
                    *(***is** *"(pc',s') $\in$ set (?step pc)")*

  from *bounded pc step* **have** *pc': "pc' < length ins"* **by** *(rule boundedD)*

  have *tkpc: "wtl (take pc ins) c 0 s0 $\neq$ $\top$"* **(is** *"?s1 $\neq$ _")* **by** *(rule wtl_take)*

```
  have s2: "wtl (take (pc+1) ins) c 0 s0 ≠ ⊤" (is "?s2 ≠ _") by (rule wtl_take)

  from wtl pc have wt_s1: "wtc c pc ?s1 ≠ ⊤" by (rule wtl_all)

  have c_Some: "∀pc t. pc < length ins ⟶ c!pc ≠ ⊥ ⟶ φ!pc = c!pc"
    by (simp add: phi_def)
  have c_None: "c!pc = ⊥ ⟹ φ!pc = ?s1" ..

  from wt_s1 pc c_None c_Some
  have inst: "wtc c pc ?s1  = wti c pc (φ!pc)"
    by (simp add: wtc split: split_if_asm)

  have "?s1 ∈ A" by (rule wtl_pres)
  with pc c_Some cert c_None
  have "φ!pc ∈ A" by (cases "c!pc = ⊥") (auto dest: cert_okD1)
  with pc pres
  have step_in_A: "snd`set (?step pc) ⊆ A" by (auto dest: pres_typeD2)

  show "s' <=_r φ!pc'"
  proof (cases "pc' = pc+1")
    case True
    with pc' cert
    have cert_in_A: "c!(pc+1) ∈ A" by (auto dest: cert_okD1)
    from True pc' have pc1: "pc+1 < length ins" by simp
    with tkpc have "?s2 = wtc c pc ?s1" by - (rule wtl_Suc)
    with inst
    have merge: "?s2 = merge c pc (?step pc) (c!(pc+1))" by (simp add: wti)
    also
    from s2 merge have "... ≠ ⊤" (is "?merge ≠ _") by simp
    with cert_in_A step_in_A
    have "?merge = (map snd [(p',t')∈?step pc. p'=pc+1] ++_f (c!(pc+1)))"
      by (rule merge_not_top_s)
    finally
    have "s' <=_r ?s2" using step_in_A cert_in_A True step
      by (auto intro: pp_ub1')
    also
    from wtl pc1 have "?s2 <=_r φ!(pc+1)" by (rule wtl_suc_pc)
    also note True [symmetric]
    finally show ?thesis by simp
  next
    case False
    from wt_s1 inst
    have "merge c pc (?step pc) (c!(pc+1)) ≠ ⊤" by (simp add: wti)
    with step_in_A
    have "∀(pc', s')∈set (?step pc). pc'≠pc+1 ⟶ s' <=_r c!pc'"
      by - (rule merge_not_top)
    with step False
    have ok: "s' <=_r c!pc'" by blast
    moreover
    from ok
    have "c!pc' = ⊥ ⟹ s' = ⊥" by simp
    moreover
    from c_Some pc'
    have "c!pc' ≠ ⊥ ⟹ φ!pc' = c!pc'" by auto
```

```
    ultimately
    show ?thesis by (cases "c!pc' = ⊥") auto
  qed
qed


lemma (in lbvs) phi_not_top:
  assumes wtl: "wtl ins c 0 s0 ≠ ⊤"
  assumes pc:  "pc < length ins"
  shows "φ!pc ≠ ⊤"
proof (cases "c!pc = ⊥")
  case False with pc
  have "φ!pc = c!pc" ..
  also from cert pc have "... ≠ ⊤" by (rule cert_okD4)
  finally show ?thesis .
next
  case True with pc
  have "φ!pc = wtl (take pc ins) c 0 s0" ..
  also from wtl have "... ≠ ⊤" by (rule wtl_take)
  finally show ?thesis .
qed

lemma (in lbvs) phi_in_A:
  assumes wtl: "wtl ins c 0 s0 ≠ ⊤"
  assumes s0:  "s0 ∈ A"
  shows "φ ∈ list (length ins) A"
proof -
  { fix x assume "x ∈ set φ"
    then obtain xs ys where "φ = xs @ x # ys"
      by (auto simp add: in_set_conv_decomp)
    then obtain pc where pc: "pc < length φ" and x: "φ!pc = x"
      by (simp add: that [of "length xs"] nth_append)

    from wtl s0 pc
    have "wtl (take pc ins) c 0 s0 ∈ A" by (auto intro!: wtl_pres)
    moreover
    from pc have "pc < length ins" by simp
    with cert have "c!pc ∈ A" ..
    ultimately
    have "φ!pc ∈ A" using pc by (simp add: phi_def)
    hence "x ∈ A" using x by simp
  }
  hence "set φ ⊆ A" ..
  thus ?thesis by (unfold list_def) simp
qed


lemma (in lbvs) phi0:
  assumes wtl: "wtl ins c 0 s0 ≠ ⊤"
  assumes 0:   "0 < length ins"
  shows "s0 <=_r φ!0"
proof (cases "c!0 = ⊥")
  case True
  with 0 have "φ!0 = wtl (take 0 ins) c 0 s0" ..
```

```
  moreover have "wtl (take 0 ins) c 0 s0 = s0" by simp
  ultimately have "φ!0 = s0" by simp
  thus ?thesis by simp
next
  case False
  with 0 have "phi!0 = c!0" ..
  moreover
  have "wtl (take 1 ins) c 0 s0 ≠ ⊤"  by (rule wtl_take)
  with 0 False
  have "s0 <=_r c!0" by (auto simp add: neq_Nil_conv wtc split: split_if_asm)
  ultimately
  show ?thesis by simp
qed


theorem (in lbvs) wtl_sound:
  assumes "wtl ins c 0 s0 ≠ ⊤"
  assumes "s0 ∈ A"
  shows "∃ts. wt_step r ⊤ step ts"
proof -
  have "wt_step r ⊤ step φ"
  proof (unfold wt_step_def, intro strip conjI)
    fix pc assume "pc < length φ"
    then obtain "pc < length ins" by simp
    show "φ!pc ≠ ⊤" by (rule phi_not_top)
    show "stable r step φ pc" by (rule wtl_stable)
  qed
  thus ?thesis ..
qed


theorem (in lbvs) wtl_sound_strong:
  assumes "wtl ins c 0 s0 ≠ ⊤"
  assumes "s0 ∈ A"
  assumes "0 < length ins"
  shows "∃ts ∈ list (length ins) A. wt_step r ⊤ step ts ∧ s0 <=_r ts!0"
proof -
  have "φ ∈ list (length ins) A" by (rule phi_in_A)
  moreover
  have "wt_step r ⊤ step φ"
  proof (unfold wt_step_def, intro strip conjI)
    fix pc assume "pc < length φ"
    then obtain "pc < length ins" by simp
    show "φ!pc ≠ ⊤" by (rule phi_not_top)
    show "stable r step φ pc" by (rule wtl_stable)
  qed
  moreover
  have "s0 <=_r φ!0" by (rule phi0)
  ultimately
  show ?thesis by fast
qed

end
```

4.19 Completeness of the LBV

theory *LBVComplete = LBVSpec + Typing_Framework:*

constdefs
  is_target :: "['s step_type, 's list, nat] ⇒ bool"
  "is_target step phi pc' ≡
    ∃pc s'. pc' ≠ pc+1 ∧ pc < length phi ∧ (pc',s') ∈ set (step pc (phi!pc))"

  make_cert :: "['s step_type, 's list, 's] ⇒ 's certificate"
  "make_cert step phi B ≡
    map (λpc. if is_target step phi pc then phi!pc else B) [0..length phi(] @ [B]"

constdefs
  list_ex :: "('a ⇒ bool) ⇒ 'a list ⇒ bool"
  "list_ex P xs ≡ ∃x ∈ set xs. P x"

lemma *[code]: "list_ex P [] = False"* **by** *(simp add: list_ex_def)*
lemma *[code]: "list_ex P (x#xs) = (P x ∨ list_ex P xs)"* **by** *(simp add: list_ex_def)*

lemma *[code]:*
  "is_target step phi pc' =
  list_ex (λpc. pc' ≠ pc+1 ∧ pc' mem (map fst (step pc (phi!pc)))) [0..length phi(]"
  apply *(simp add: list_ex_def is_target_def set_mem_eq)*
  apply *force*
  done

locale *(***open***) 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 *(unfold cert_ok_def, intro strip conjI)*
  note *[simp] = make_cert_def cert_def nth_append*

  show *"c!length φ = ⊥"* **by** *simp*

  fix *pc* **assume** *pc: "pc < length φ"*
  from *pc phi B_A* **show** *"c!pc ∈ A"* **by** *simp*
  from *pc phi B_neq_T* **show** *"c!pc ≠ ⊤"* **by** *simp*
qed

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'"
  by (auto simp add: cert_def make_cert_def nth_append is_target_def)


lemma (in lbvc) cert_approx [intro?]:
  "⟦ pc < length φ; c!pc ≠ ⊥ ⟧
  ⟹ c!pc = φ!pc"
  by (auto simp add: cert_def make_cert_def nth_append)


lemma (in lbv) le_top [simp, intro]:
  "x <=_r ⊤"
  by (insert top) simp


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-
  have "?s1 = ⊤ ⟹ ?thesis" by simp
  moreover {
    assume merge: "?s1 ≠ T"
    from x ss1 have "?s1 =
      (if ∀(pc', s')∈set ss1. pc' ≠ pc + 1 ⟶ s' <=_r c!pc'
      then (map snd [(p', t')∈ss1 . p'=pc+1]) ++_f x
      else ⊤)"
      by (rule merge_def)
    with merge obtain
      app: "∀(pc',s')∈set ss1. pc' ≠ pc+1 ⟶ s' <=_r c!pc'"
           (is "?app ss1") and
      sum: "(map snd [(p',t')∈ss1 . p' = pc+1] ++_f x) = ?s1"
           (is "?map ss1 ++_f x = _" is "?sum ss1 = _")
      by (simp split: split_if_asm)
    from app less
    have "?app ss2" by (blast dest: trans_r lesub_step_typeD)
    moreover {
      from ss1 have map1: "set (?map ss1) ⊆ A" by auto
      with x have "?sum ss1 ∈ A" by (auto intro!: plusplus_closed)
      with sum have "?s1 ∈ A" by simp
      moreover
      have mapD: "⋀x ss. x ∈ set (?map ss) ⟹ ∃p. (p,x) ∈ set ss ∧ p=pc+1" by auto
      from x map1
      have "∀x ∈ set (?map ss1). x <=_r ?sum ss1"
        by clarify (rule pp_ub1)
      with sum have "∀x ∈ set (?map ss1). x <=_r ?s1" by simp
      with less have "∀x ∈ set (?map ss2). x <=_r ?s1"
        by (fastsimp dest!: mapD lesub_step_typeD intro: trans_r)
      moreover
```

```
      from map1 x have "x <=_r (?sum ss1)" by (rule pp_ub2)
      with sum have "x <=_r ?s1" by simp
      moreover
      from ss2 have "set (?map ss2) ⊆ A" by auto
      ultimately
      have "?sum ss2 <=_r ?s1" using x by - (rule pp_lub)
    }
    moreover
    from x ss2 have
      "?s2 =
      (if ∀(pc', s')∈set ss2. pc' ≠ pc + 1 ⟶ s' <=_r c!pc'
      then map snd [(p', t')∈ss2 . p' = pc + 1] ++_f x
      else ⊤)"
      by (rule merge_def)
    ultimately have ?thesis by simp
  }
  ultimately show ?thesis by (cases "?s1 = ⊤") auto
qed


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 -
  from mono s2 have "step pc s2 <=|r| step pc s1" by - (rule monoD)
  moreover
  from pc cert have "c!Suc pc ∈ A" by - (rule cert_okD3)
  moreover
  from pres s1 pc
  have "snd`set (step pc s1) ⊆ A" by (rule pres_typeD2)
  moreover
  from pres s2 pc
  have "snd`set (step pc s2) ⊆ A" by (rule pres_typeD2)
  ultimately
  show ?thesis by (simp add: wti merge_mono)
qed

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 (cases "c!pc = ⊥")
  case True
  moreover have "wti c pc s2 <=_r wti c pc s1" by (rule wti_mono)
  ultimately show ?thesis by (simp add: wtc)
next
  case False
  have "?s1' = ⊤ ⟹ ?thesis" by simp
  moreover {
```

```
    assume "?s1' ≠ ⊤"
    with False have c: "s1 <=_r c!pc" by (simp add: wtc split: split_if_asm)
    with less have "s2 <=_r c!pc" ..
    with False c have ?thesis by (simp add: wtc)
  }
  ultimately show ?thesis by (cases "?s1' = ⊤") auto
qed


lemma (in lbv) top_le_conv [simp]:
  "⊤ <=_r x = (x = ⊤)"
  by (insert semilat) (simp add: top top_le_conv)

lemma (in lbv) neq_top [simp, elim]:
  "⟦ x <=_r y; y ≠ ⊤ ⟧ ⟹ x ≠ ⊤"
  by (cases "x = T") auto


lemma (in lbvc) stable_wti:
  assumes stable:  "stable r step φ pc"
  assumes pc:      "pc < length φ"
  shows "wti c pc (φ!pc) ≠ ⊤"
proof -
  let ?step = "step pc (φ!pc)"
  from stable
  have less: "∀(q,s')∈set ?step. s' <=_r φ!q" by (simp add: stable_def)

  from cert pc
  have cert_suc: "c!Suc pc ∈ A" by - (rule cert_okD3)
  moreover
  from phi pc have "φ!pc ∈ A" by simp
  with pres pc
  have stepA: "snd`set ?step ⊆ A" by - (rule pres_typeD2)
  ultimately
  have "merge c pc ?step (c!Suc pc) =
    (if ∀(pc',s')∈set ?step. pc'≠pc+1 ⟶ s' <=_r c!pc'
    then map snd [(p',t')∈?step.p'=pc+1] ++_f c!Suc pc
    else ⊤)" by (rule merge_def)
  moreover {
    fix pc' s' assume s': "(pc', s') ∈ set ?step" and suc_pc: "pc' ≠ pc+1"
    with less have "s' <=_r φ!pc'" by auto
    also
    from bounded pc s' have "pc' < length φ" by (rule boundedD)
    with s' suc_pc pc have "c!pc' = φ!pc'" ..
    hence "φ!pc' = c!pc'" ..
    finally have "s' <=_r c!pc'" .
  } hence "∀(pc',s')∈set ?step. pc'≠pc+1 ⟶ s' <=_r c!pc'" by auto
  moreover
  from pc have "Suc pc = length φ ∨ Suc pc < length φ" by auto
  hence "map snd [(p',t')∈?step.p'=pc+1] ++_f c!Suc pc ≠ ⊤"
        (is "?map ++_f _ ≠ _")
  proof (rule disjE)
    assume pc': "Suc pc = length φ"
    with cert have "c!Suc pc = ⊥" by (simp add: cert_okD2)
```

```
    moreover
    from pc' bounded pc
    have "∀(p',t')∈set ?step. p'≠pc+1" by clarify (drule boundedD, auto)
    hence "[(p',t')∈?step.p'=pc+1] = []" by (blast intro: filter_False)
    hence "?map = []" by simp
    ultimately show ?thesis by (simp add: B_neq_T)
  next
    assume pc': "Suc pc < length φ"
    from pc' phi have "φ!Suc pc ∈ A" by simp
    moreover note cert_suc
    moreover from stepA
    have "set ?map ⊆ A" by auto
    moreover
    have "⋀s. s ∈ set ?map ⟹ ∃t. (Suc pc, t) ∈ set ?step" by auto
    with less have "∀s' ∈ set ?map. s' <=_r φ!Suc pc" by auto
    moreover
    from pc' have "c!Suc pc <=_r φ!Suc pc"
      by (cases "c!Suc pc = ⊥") (auto dest: cert_approx)
    ultimately
    have "?map ++_f c!Suc pc <=_r φ!Suc pc" by (rule pp_lub)
    moreover
    from pc' phi have "φ!Suc pc ≠ ⊤" by simp
    ultimately
    show ?thesis by auto
  qed
  ultimately
  have "merge c pc ?step (c!Suc pc) ≠ ⊤" by simp
  thus ?thesis by (simp add: wti)
qed

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 -
  let ?step = "step pc (φ!pc)"

  from stable
  have less: "∀(q,s')∈set ?step. s' <=_r φ!q" by (simp add: stable_def)

  from suc_pc have pc: "pc < length φ" by simp
  with cert have cert_suc: "c!Suc pc ∈ A" by - (rule cert_okD3)
  moreover
  from phi pc have "φ!pc ∈ A" by simp
  with pres pc have stepA: "snd`set ?step ⊆ A" by - (rule pres_typeD2)
  moreover
  from stable pc have "?wti ≠ ⊤" by (rule stable_wti)
  hence "merge c pc ?step (c!Suc pc) ≠ ⊤" by (simp add: wti)
  ultimately
  have "merge c pc ?step (c!Suc pc) =
    map snd [(p',t')∈?step.p'=pc+1] ++_f c!Suc pc" by (rule merge_not_top_s)
  hence "?wti = ..." (is "_ = (?map ++_f _)" is "_ = ?sum") by (simp add: wti)
  also {
    from suc_pc phi have "φ!Suc pc ∈ A" by simp
```

```
    moreover note cert_suc
    moreover from stepA have "set ?map ⊆ A" by auto
    moreover
    have "⋀s. s ∈ set ?map ⟹ ∃t. (Suc pc, t) ∈ set ?step" by auto
    with less have "∀s' ∈ set ?map. s' <=_r φ!Suc pc" by auto
    moreover
    from suc_pc have "c!Suc pc <=_r φ!Suc pc"
      by (cases "c!Suc pc = ⊥") (auto dest: cert_approx)
    ultimately
    have "?sum <=_r φ!Suc pc" by (rule pp_lub)
  }
  finally show ?thesis .
qed

lemma (in lbvc) stable_wtc:
  assumes stable:  "stable r step phi pc"
  assumes pc:      "pc < length φ"
  shows "wtc c pc (φ!pc) ≠ ⊤"
proof -
  have wti: "wti c pc (φ!pc) ≠ ⊤" by (rule stable_wti)
  show ?thesis
  proof (cases "c!pc = ⊥")
    case True with wti show ?thesis by (simp add: wtc)
  next
    case False
    with pc have "c!pc = φ!pc" ..
    with False wti show ?thesis by (simp add: wtc)
  qed
qed

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 (cases "c!pc = ⊥")
  case True
  moreover have "wti c pc (φ!pc) <=_r φ!Suc pc" by (rule wti_less)
  ultimately show ?thesis by (simp add: wtc)
next
  case False
  from suc_pc have pc: "pc < length φ" by simp
  hence "?wtc ≠ ⊤" by - (rule stable_wtc)
  with False have "?wtc = wti c pc (c!pc)"
    by (unfold wtc) (simp split: split_if_asm)
  also from pc False have "c!pc = φ!pc" ..
  finally have "?wtc = wti c pc (φ!pc)" .
  also have "wti c pc (φ!pc) <=_r φ!Suc pc" by (rule wti_less)
  finally show ?thesis .
qed


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 (induct ls)
  fix pc s assume "s≠⊤" thus "?wtl [] pc s ≠ ⊤" by simp
next
  fix pc s i ls
  assume "⋀pc s. pc+length ls=length φ ⟹ s <=_r φ!pc ⟹ s ∈ A ⟹ s≠⊤ ⟹
                 ?wtl ls pc s ≠ ⊤"
  moreover
  assume pc_l: "pc + length (i#ls) = length φ"
  hence suc_pc_l: "Suc pc + length ls = length φ" by simp
  ultimately
  have IH: "⋀s. s <=_r φ!Suc pc ⟹ s ∈ A ⟹ s ≠ ⊤ ⟹ ?wtl ls (Suc pc) s ≠ ⊤" .

  from pc_l obtain pc: "pc < length φ" by simp
  with wt_step have stable: "stable r step φ pc" by (simp add: wt_step_def)
  moreover
  assume s_phi: "s <=_r φ!pc"
  ultimately
  have wt_phi: "wtc c pc (φ!pc) ≠ ⊤" by - (rule stable_wtc)

  from phi pc have phi_pc: "φ!pc ∈ A" by simp
  moreover
  assume s: "s ∈ A"
  ultimately
  have wt_s_phi: "wtc c pc s <=_r wtc c pc (φ!pc)" using s_phi by - (rule wtc_mono)
  with wt_phi have wt_s: "wtc c pc s ≠ ⊤" by simp
  moreover
  assume s: "s ≠ ⊤"
  ultimately
  have "ls = [] ⟹ ?wtl (i#ls) pc s ≠ ⊤" by simp
  moreover {
    assume "ls ≠ []"
    with pc_l have suc_pc: "Suc pc < length φ" by (auto simp add: neq_Nil_conv)
    with stable have "wtc c pc (phi!pc) <=_r φ!Suc pc" by (rule wtc_less)
    with wt_s_phi have "wtc c pc s <=_r φ!Suc pc" by (rule trans_r)
    moreover
    from cert suc_pc have "c!pc ∈ A" "c!(pc+1) ∈ A"
      by (auto simp add: cert_ok_def)
    with pres have "wtc c pc s ∈ A" by (rule wtc_pres)
    ultimately
    have "?wtl ls (Suc pc) (wtc c pc s) ≠ ⊤" using IH wt_s by blast
    with s wt_s have "?wtl (i#ls) pc s ≠ ⊤" by simp
  }
  ultimately show "?wtl (i#ls) pc s ≠ ⊤" by (cases ls) blast+
qed


theorem (in lbvc) wtl_complete:
  assumes "wt_step r ⊤ step φ"
  assumes "s <=_r φ!0" and "s ∈ A" and "s ≠ ⊤" and "length ins = length phi"
  shows "wtl ins c 0 s ≠ ⊤"
proof -
  have "0+length ins = length phi" by simp
```

thus *?thesis* **by** - *(rule wt_step_wtl_lemma)*
qed

end

4.20 LBV for the JVM

theory *LBVJVM = LBVCorrect + LBVComplete + Typing_Framework_JVM:*

types *prog_cert = "cname ⇒ sig ⇒ state list"*

constdefs

```
  check_cert :: "jvm_prog ⇒ nat ⇒ nat ⇒ nat ⇒ state list ⇒ bool"
  "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"

  lbvjvm :: "jvm_prog ⇒ nat ⇒ nat ⇒ ty ⇒ exception_table ⇒
             state list ⇒ instr list ⇒ state ⇒ state"
  "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"

  wt_lbv :: "jvm_prog ⇒ cname ⇒ ty list ⇒ ty ⇒ nat ⇒ nat ⇒
             exception_table ⇒ state list ⇒ instr list ⇒ bool"
  "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)"

  wt_jvm_prog_lbv :: "jvm_prog ⇒ prog_cert ⇒ bool"
  "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"

  mk_cert :: "jvm_prog ⇒ nat ⇒ ty ⇒ exception_table ⇒ instr list
              ⇒ method_type ⇒ state list"
  "mk_cert G maxs rT et bs phi ≡ make_cert (exec G maxs rT et bs) (map OK phi) (OK None)"

  prg_cert :: "jvm_prog ⇒ prog_type ⇒ prog_cert"
  "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)"
  by (auto simp add: wt_method_def wt_app_eff_def wt_instr_def lesub_def
           dest: check_bounded_is_bounded boundedD)


lemma check_certD:
  "check_cert G mxs mxr n cert ⟹ cert_ok cert n Err (OK None) (states G mxs mxr)"
  apply (unfold cert_ok_def check_cert_def check_types_def)
  apply (auto simp add: list_all_ball)
  done


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 -
  let ?step = "exec G mxs rT et ins"
  let ?r    = "JVMType.le G mxs mxr"
  let ?f    = "JVMType.sup G mxs mxr"
  let ?A    = "states G mxs mxr"

  have "semilat (JVMType.sl G mxs mxr)" by (rule semilat_JVM_slI)
  hence "semilat (?A, ?r, ?f)" by (unfold sl_triple_conv)
  moreover
  have "top ?r Err"  by (simp add: JVM_le_unfold)
  moreover
  have "Err ∈ ?A" by (simp add: JVM_states_unfold)
  moreover
  have "bottom ?r (OK None)"
    by (simp add: JVM_le_unfold bottom_def)
  moreover
  have "OK None ∈ ?A" by (simp add: JVM_states_unfold)
  moreover
  from lbv
  have "bounded ?step (length ins)"
    by (clarsimp simp add: wt_lbv_def exec_def)
       (intro bounded_lift check_bounded_is_bounded)
  moreover
  from lbv
  have "cert_ok cert (length ins) Err (OK None) ?A"
    by (unfold wt_lbv_def) (auto dest: check_certD)
  moreover
  have "pres_type ?step (length ins) ?A" by (rule exec_pres_type)
  moreover
```

```
  let ?start = "OK (Some ([],(OK (Class C))#(map OK pTs)@(replicate mxl Err)))"
  from lbv
  have "wtl_inst_list ins cert ?f ?r Err (OK None) ?step 0 ?start ≠ Err"
    by (simp add: wt_lbv_def lbvjvm_def)
  moreover
  from C pTs have "?start ∈ ?A"
    by (unfold JVM_states_unfold) (auto intro: list_appendI, force)
  moreover
  from lbv have "0 < length ins" by (simp add: wt_lbv_def)
  ultimately
  show ?thesis by (rule lbvs.wtl_sound_strong)
qed

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 -
  let ?mxr   = "1 + length pTs + mxl"
  let ?step  = "exec G mxs rT et ins"
  let ?r     = "JVMType.le G mxs ?mxr"
  let ?f     = "JVMType.sup G mxs ?mxr"
  let ?A     = "states G mxs ?mxr"
  let ?start = "OK (Some ([],(OK (Class C))#(map OK pTs)@(replicate mxl Err)))"

  from lbv have l: "ins ≠ []" by (simp add: wt_lbv_def)
  moreover
  from wf lbv C pTs
  obtain phi where
    list:  "phi ∈ list (length ins) ?A" and
    step:  "wt_step ?r Err ?step phi" and
    start: "?start <=_?r phi!0"
    by (blast dest: wt_lbv_wt_step)
  from list have [simp]: "length phi = length ins" by simp
  have "length (map ok_val phi) = length ins" by simp
  moreover
  from l have 0: "0 < length phi" by simp
  with step obtain phi0 where "phi!0 = OK phi0"
    by (unfold wt_step_def) blast
  with start 0
  have "wt_start G C pTs mxl (map ok_val phi)"
    by (simp add: wt_start_def JVM_le_Err_conv lesub_def)
  moreover
  from lbv  have chk_bounded: "check_bounded ins et"
    by (simp add: wt_lbv_def)
  moreover {
    from list
    have "check_types G mxs ?mxr phi"
      by (simp add: check_types_def)
    also from step
    have [symmetric]: "map OK (map ok_val phi) = phi"
```

```
      by (auto intro!: map_id simp add: wt_step_def)
    finally have "check_types G mxs ?mxr (map OK (map ok_val phi))" .
  }
  moreover {
    let ?app = "λpc. app (ins!pc) G mxs rT pc et"
    let ?eff = "λpc. eff (ins!pc) G pc et"

    from chk_bounded
    have "bounded (err_step (length ins) ?app ?eff) (length ins)"
      by (blast dest: check_bounded_is_bounded boundedD intro: bounded_err_stepI)
    moreover
    from step
    have "wt_err_step (sup_state_opt G) ?step phi"
      by (simp add: wt_err_step_def JVM_le_Err_conv)
    ultimately
    have "wt_app_eff (sup_state_opt G) ?app ?eff (map ok_val phi)"
      by (auto intro: wt_err_imp_wt_app_eff simp add: exec_def)
  }
  ultimately
  have "wt_method G C pTs rT mxs mxl ins et (map ok_val phi)"
    by - (rule wt_method_def2 [THEN iffD2], simp)
  thus ?thesis ..
qed


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 -
  let ?mxr  = "1 + length pTs + mxl"
  let ?step = "exec G mxs rT et ins"
  let ?app  = "λpc. app (ins!pc) G mxs rT pc et"
  let ?eff  = "λpc. eff (ins!pc) G pc et"
  let ?r    = "JVMType.le G mxs ?mxr"
  let ?f    = "JVMType.sup G mxs ?mxr"
  let ?A    = "states G mxs ?mxr"
  let ?phi  = "map OK phi"
  let ?cert = "make_cert ?step ?phi (OK None)"

  from wt obtain
    0:          "0 < length ins" and
    length:     "length ins = length ?phi" and
    ck_bounded: "check_bounded ins et" and
    ck_types:   "check_types G mxs ?mxr ?phi" and
    wt_start:   "wt_start G C pTs mxl phi" and
    app_eff:    "wt_app_eff (sup_state_opt G) ?app ?eff phi"
    by (simp (asm_lr) add: wt_method_def2)
```

```
have "semilat (JVMType.sl G mxs ?mxr)" by (rule semilat_JVM_slI)
hence "semilat (?A, ?r, ?f)" by (unfold sl_triple_conv)
moreover
have "top ?r Err"  by (simp add: JVM_le_unfold)
moreover
have "Err ∈ ?A" by (simp add: JVM_states_unfold)
moreover
have "bottom ?r (OK None)"
  by (simp add: JVM_le_unfold bottom_def)
moreover
have "OK None ∈ ?A" by (simp add: JVM_states_unfold)
moreover
from ck_bounded
have bounded: "bounded ?step (length ins)"
  by (clarsimp simp add: exec_def)
     (intro bounded_lift check_bounded_is_bounded)
with wf
have "mono ?r ?step (length ins) ?A" by (rule exec_mono)
hence "mono ?r ?step (length ?phi) ?A" by (simp add: length)
moreover
have "pres_type ?step (length ins) ?A" by (rule exec_pres_type)
hence "pres_type ?step (length ?phi) ?A" by (simp add: length)
moreover
from ck_types
have "set ?phi ⊆ ?A" by (simp add: check_types_def)
hence "∀pc. pc < length ?phi ⟶ ?phi!pc ∈ ?A ∧ ?phi!pc ≠ Err" by auto
moreover
from bounded
have "bounded (exec G mxs rT et ins) (length ?phi)" by (simp add: length)
moreover
have "OK None ≠ Err" by simp
moreover
from bounded length app_eff
have "wt_err_step (sup_state_opt G) ?step ?phi"
  by (auto intro: wt_app_eff_imp_wt_err simp add: exec_def)
hence "wt_step ?r Err ?step ?phi"
  by (simp add: wt_err_step_def JVM_le_Err_conv)
moreover
let ?start = "OK (Some ([],(OK (Class C))#(map OK pTs)@(replicate mxl Err)))"
from 0 length have "0 < length phi" by auto
hence "?phi!0 = OK (phi!0)" by simp
with wt_start have "?start <=_?r ?phi!0"
  by (clarsimp simp add: wt_start_def lesub_def JVM_le_Err_conv)
moreover
from C pTs have "?start ∈ ?A"
  by (unfold JVM_states_unfold) (auto intro: list_appendI, force)
moreover
have "?start ≠ Err" by simp
moreover
note length
ultimately
have "wtl_inst_list ins ?cert ?f ?r Err (OK None) ?step 0 ?start ≠ Err"
  by (rule lbvc.wtl_complete)
moreover
```

```
  from 0 length have "phi ≠ []" by auto
  moreover
  from ck_types
  have "check_types G mxs ?mxr ?cert"
    by (auto simp add: make_cert_def check_types_def JVM_states_unfold)
  moreover
  note ck_bounded 0 length
  ultimately
  show ?thesis
    by (simp add: wt_lbv_def lbvjvm_def mk_cert_def
      check_cert_def make_cert_def nth_append)
qed



theorem jvm_lbv_correct:
  "wt_jvm_prog_lbv G Cert ⟹ ∃Phi. wt_jvm_prog G Phi"
proof -
  let ?Phi = "λC sig. let (C,rT,(maxs,maxl,ins,et)) = the (method (G,C) sig) in
              SOME phi. wt_method G C (snd sig) rT maxs maxl ins et phi"

  assume "wt_jvm_prog_lbv G Cert"
  hence "wt_jvm_prog G ?Phi"
    apply (unfold wt_jvm_prog_def wt_jvm_prog_lbv_def)
    apply (erule jvm_prog_lift)
    apply (auto dest: wt_lbv_wt_method intro: someI)
    done
  thus ?thesis by blast
qed

theorem jvm_lbv_complete:
  "wt_jvm_prog G Phi ⟹ wt_jvm_prog_lbv G (prg_cert G Phi)"
  apply (unfold wt_jvm_prog_def wt_jvm_prog_lbv_def)
  apply (erule jvm_prog_lift)
  apply (auto simp add: prg_cert_def intro wt_method_wt_lbv)
  done

end
```

4.21 BV Type Safety Invariant

theory *Correct = BVSpec + JVMExec:*

constdefs

```
  approx_val :: "[jvm_prog,aheap,val,ty err] ⇒ bool"
  "approx_val G h v any == case any of Err ⇒ True | OK T ⇒ G,h⊢v::≼T"

  approx_loc :: "[jvm_prog,aheap,val list,locvars_type] ⇒ bool"
  "approx_loc G hp loc LT == list_all2 (approx_val G hp) loc LT"

  approx_stk :: "[jvm_prog,aheap,opstack,opstack_type] ⇒ bool"
  "approx_stk G hp stk ST == approx_loc G hp stk (map OK ST)"

  correct_frame  :: "[jvm_prog,aheap,state_type,nat,bytecode] ⇒ frame ⇒ bool"
  "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"
```

consts

```
 correct_frames  :: "[jvm_prog,aheap,prog_type,ty,sig,frame list] ⇒ bool"
```

primrec

```
"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))))"
```

constdefs

```
 correct_state :: "[jvm_prog,prog_type,jvm_state] ⇒ bool"
                  ("_,_ |-JVM _ [ok]"  [51,51] 50)
"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 = []"


syntax (xsymbols)
 correct_state :: "[jvm_prog,prog_type,jvm_state] ⇒ bool"
                  ("_,_ ⊢JVM _ √"  [51,51] 50)


lemma sup_ty_opt_OK:
  "(G ⊢ X <=o (OK T')) = (∃T. X = OK T ∧ G ⊢ T ≼ T')"
  apply (cases X)
  apply auto
  done
```

4.21.1 approx-val

```
lemma approx_val_Err [simp,intro!]:
  "approx_val G hp x Err"
  by (simp add: approx_val_def)

lemma approx_val_OK [iff]:
  "approx_val G hp x (OK T) = (G,hp ⊢ x ::≼ T)"
  by (simp add: approx_val_def)

lemma approx_val_Null [simp,intro!]:
  "approx_val G hp Null (OK (RefT x))"
  by (auto simp add: approx_val_def)

lemma approx_val_sup_heap:
  "⟦ approx_val G hp v T; hp ≤| hp' ⟧ ⟹ approx_val G hp' v T"
  by (cases T) (blast intro: conf_hext)+

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"
  by (cases v, auto simp add: obj_ty_def conf_def)

lemma approx_val_widen:
  "⟦ approx_val G hp v T; G ⊢ T <=o T'; wf_prog wt G ⟧
  ⟹ approx_val G hp v T'"
  by (cases T', auto simp add: sup_ty_opt_OK intro: conf_widen)
```

4.21.2 approx-loc

```
lemma approx_loc_Nil [simp,intro!]:
  "approx_loc G hp [] []"
  by (simp add: approx_loc_def)

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)"
by (simp add: approx_loc_def)

lemma approx_loc_nth:
  "⟦ approx_loc G hp loc LT; n < length LT ⟧
  ⟹ approx_val G hp (loc!n) (LT!n)"
  by (simp add: approx_loc_def list_all2_conv_all_nth)

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'"
  apply (drule approx_loc_nth, assumption)
  apply simp
  apply (erule conf_widen, assumption+)
  done

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)))"
  by (simp add: approx_loc_def list_all2_conv_all_nth)

lemma approx_loc_sup_heap:
  "⟦ approx_loc G hp loc LT; hp ≤| hp' ⟧
  ⟹ approx_loc G hp' loc LT"
  apply (clarsimp simp add: approx_loc_conv_all_nth)
  apply (blast intro: approx_val_sup_heap)
  done

lemma approx_loc_widen:
  "⟦ approx_loc G hp loc LT; G ⊢ LT <=l LT'; wf_prog wt G ⟧
  ⟹ approx_loc G hp loc LT'"
apply (unfold Listn.le_def lesub_def sup_loc_def)
apply (simp (no_asm_use) only: list_all2_conv_all_nth approx_loc_conv_all_nth)
apply (simp (no_asm_simp))
apply clarify
apply (erule allE, erule impE)
 apply simp
apply (erule approx_val_widen)
 apply simp
apply assumption
done

lemma loc_widen_Err [dest]:
  "⋀XT. G ⊢ replicate n Err <=l XT ⟹ XT = replicate n Err"
  by (induct n) auto

lemma approx_loc_Err [iff]:
  "approx_loc G hp (replicate n v) (replicate n Err)"
  by (induct n) auto

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])"
apply (unfold approx_loc_def list_all2_def)
```

apply *(auto dest: subsetD [OF set_update_subset_insert] simp add: zip_update)*
done

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)"
apply *(unfold approx_loc_def list_all2_def)*
apply *(simp cong: conj_cong)*
apply *blast*
done

4.21.3 approx-stk

lemma *approx_stk_rev_lem:*
"approx_stk G hp (rev s) (rev t) = approx_stk G hp s t"
apply *(unfold approx_stk_def approx_loc_def)*
apply *(simp add: rev_map [THEN sym])*
done

lemma *approx_stk_rev:*
"approx_stk G hp (rev s) t = approx_stk G hp s (rev t)"
by *(auto intro: subst [OF approx_stk_rev_lem])*

lemma *approx_stk_sup_heap:*
"⟦ approx_stk G hp stk ST; hp ≤| hp' ⟧ ⟹ approx_stk G hp' stk ST"
by *(auto intro: approx_loc_sup_heap simp add: approx_stk_def)*

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'"
by *(auto elim: approx_loc_widen simp add: approx_stk_def)*

lemma *approx_stk_Nil [iff]:*
"approx_stk G hp [] []"
by *(simp add: approx_stk_def)*

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)"
by *(simp add: approx_stk_def)*

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')"
by *(simp add: list_all2_Cons2 approx_stk_def approx_loc_def)*

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')"
by *(simp add: list_all2_append2 approx_stk_def approx_loc_def)*

lemma *approx_stk_all_widen:*

```
  "⟦ approx_stk G hp stk ST; ∀x ∈ set (zip ST ST'). x ∈ widen G; length ST = length ST';
wf_prog wt G ⟧
  ⟹ approx_stk G hp stk ST'"
apply (unfold approx_stk_def)
apply (clarsimp simp add: approx_loc_conv_all_nth all_set_conv_all_nth)
apply (erule allE, erule impE, assumption)
apply (erule allE, erule impE, assumption)
apply (erule conf_widen, assumption+)
done
```

4.21.4 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))√"
  by (simp add: oconf_def lconf_def)

lemma oconf_newref:
  "⟦hp oref = None; G,hp ⊢ obj √; G,hp ⊢ obj' √⟧ ⟹ G,hp(oref↦obj') ⊢ obj √"
  apply (unfold oconf_def lconf_def)
  apply simp
  apply (blast intro: conf_hext hext_new)
  done

lemma oconf_heap_update:
  "⟦ hp a = Some obj'; obj_ty obj' = obj_ty obj''; G,hp⊢obj√ ⟧
  ⟹ G,hp(a↦obj'')⊢obj√"
  apply (unfold oconf_def lconf_def)
  apply (fastsimp intro: approx_val_heap_update)
  done
```

4.21.5 hconf

```
lemma hconf_newref:
  "⟦ hp oref = None; G⊢h hp√; G,hp⊢obj√ ⟧ ⟹ G⊢h hp(oref↦obj)√"
  apply (simp add: hconf_def)
  apply (fast intro: oconf_newref)
  done

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)))√"
  apply (simp add: hconf_def)
  apply (fastsimp intro: oconf_heap_update oconf_field_update
                  simp add: obj_ty_def)
  done
```

4.21.6 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))))"
  apply (unfold preallocated_def)
```

```
  apply (rule allI)
  apply (erule_tac x=x in allE)
  apply simp
  apply (rule ccontr)
  apply (unfold hconf_def)
  apply (erule allE, erule allE, erule impE, assumption)
  apply (unfold oconf_def lconf_def)
  apply (simp del: split_paired_All)
  done


lemma
  assumes none: "hp oref = None" and alloc: "preallocated hp"
  shows preallocated_newref: "preallocated (hp(oref↦obj))"
proof (cases oref)
  case (XcptRef x)
  with none alloc have "False" by (auto elim: preallocatedE [of _ x])
  thus ?thesis ..
next
  case (Loc l)
  with alloc show ?thesis by (simp add: preallocated_def)
qed
```

4.21.7 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"
apply (induct frs)
 apply simp
apply clarify
apply (simp (no_asm_use))
apply clarify
apply (unfold correct_frame_def)
apply (simp (no_asm_use))
apply clarify
apply (intro exI conjI)
    apply assumption+
   apply (erule approx_stk_sup_heap)
   apply (erule hext_upd_obj)
  apply (erule approx_loc_sup_heap)
  apply (erule hext_upd_obj)
 apply assumption+
apply blast
done

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"
apply (induct frs)
 apply simp
apply clarify
apply (simp (no_asm_use))
apply clarify
apply (unfold correct_frame_def)
apply (simp (no_asm_use))
apply clarify
apply (intro exI conjI)
    apply assumption+
   apply (erule approx_stk_sup_heap)
   apply (erule hext_new)
  apply (erule approx_loc_sup_heap)
  apply (erule hext_new)
 apply assumption+
apply blast
done

end
```

4.22 BV Type Safety Proof

theory *BVSpecTypeSafe = Correct:*

This theory contains proof that the specification of the bytecode verifier only admits type safe programs.

4.22.1 Preliminaries

Simp and intro setup for the type safety proof:

lemmas *defs1 = 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"
apply *(unfold correct_state_def Let_def correct_frame_def)*
apply *simp*
apply *(blast intro: wt_jvm_prog_impl_wt_instr)*
done

4.22.2 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))"
by *(cases i, auto simp add: split_beta split: split_if_asm)*

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 *(induct et)*
show *"PROP ?P []"*
by *simp*

fix *e es*

```
  assume IH: "PROP ?P es"
  assume match: "?match (e#es)"

  obtain start_pc end_pc handler_pc catch_type where
    e [simp]: "e = (start_pc, end_pc, handler_pc, catch_type)"
    by (cases e)

  from IH match
  show "?match_any (e#es)"
  proof (cases "match_exception_entry G xcpt pc e")
    case False
    with match
    have "match_exception_table G xcpt pc es = Some pc'" by simp
    with IH
    obtain C where
      set: "C ∈ set (match_any G pc es)" and
      C:   "G ⊢ xcpt ⪯C C" and
      m:   "match_exception_table G C pc es = Some pc'" by blast

    from set
    have "C ∈ set (match_any G pc (e#es))" by simp
    moreover
    from False C
    have "¬ match_exception_entry G C pc e"
      by - (erule contrapos_nn,
            auto simp add: match_exception_entry_def elim: rtrancl_trans)
    with m
    have "match_exception_table G C pc (e#es) = Some pc'" by simp
    moreover note C
    ultimately
    show ?thesis by blast
  next
    case True with match
    have "match_exception_entry G catch_type pc e"
      by (simp add: match_exception_entry_def)
    moreover
    from True match
    obtain
      "start_pc ≤ pc"
      "pc < end_pc"
      "G ⊢ xcpt ⪯C catch_type"
      "handler_pc = pc'"
      by (simp add: match_exception_entry_def)
    ultimately
    show ?thesis by auto
  qed
qed
```

lemma *match_et_imp_match:*
"match_exception_table G (Xcpt X) pc et = Some handler
⟹ match G X pc et = [Xcpt X]"
apply *(simp add: match_some_entry)*
apply *(induct et)*
apply *(auto split: split_if_asm)*
done

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 *(induct frs)*
— the base case is trivial, as it should be
show *"?correct (?find [])"* **by** *(simp add: correct_state_def)*

— we will need both forms *wt_jvm_prog* and *wf_prog* later
assume *wt: ?wt*
then obtain *mb* **where** *wf: "wf_prog mb G"* **by** *(simp add: wt_jvm_prog_def)*

— these two don't change in the induction:
assume *adr: ?adr*
assume *hp: ?hp*

— the assumption for the cons case:
fix *f f' frs'*
assume *cr: "?correct (None, hp, f#f'#frs')"*

— the induction hypothesis as produced by Isabelle, immediatly simplified with the fixed assumptions above
assume *"⋀f. ⟦ ?wt; ?adr; ?hp; ?correct (None, hp, f#frs') ⟧ ⟹ ?correct (?find frs')"*

with *wt adr hp*
have *IH: "⋀f. ?correct (None, hp, f#frs') ⟹ ?correct (?find frs')"* **by** *blast*

from *cr*
have *cr': "?correct (None, hp, f'#frs')"* **by** *(auto simp add: correct_state_def)*

obtain *stk loc C sig pc* **where** *f' [simp]: "f' = (stk,loc,C,sig,pc)"*
by *(cases f')*

from *cr*
obtain *rT maxs maxl ins et* **where**

```
    meth: "method (G,C) sig = Some (C,rT,maxs,maxl,ins,et)"
    by (simp add: correct_state_def, blast)

  hence [simp]: "ex_table_of (snd (snd (the (method (G, C) sig)))) = et"
    by simp

  show "?correct (?find (f'#frs'))"
  proof (cases "match_exception_table G (cname_of hp xcp) pc et")
    case None
    with cr' IH
    show ?thesis by simp
  next
    fix handler_pc
    assume match: "match_exception_table G (cname_of hp xcp) pc et = Some handler_pc"
    (is "?match (cname_of hp xcp) = _")

    from wt meth cr' [simplified]
    have wti: "wt_instr (ins!pc) G rT (phi C sig) maxs (length ins) et pc"
      by (rule wt_jvm_prog_impl_wt_instr_cor)

    from cr meth
    obtain C' mn pts ST LT where
      ins: "ins!pc = Invoke C' mn pts" (is "_ = ?i") and
      phi: "phi C sig ! pc = Some (ST, LT)"
      by (simp add: correct_state_def) blast

    from match
    obtain D where
      in_any: "D ∈ set (match_any G pc et)" and
      D:      "G ⊢ cname_of hp xcp ⪯C D" and
      match': "?match D = Some handler_pc"
      by (blast dest: in_match_any)

    from ins wti phi have
      "∀D∈set (match_any G pc et). the (?match D) < length ins ∧
      G ⊢ Some ([Class D], LT) <=' phi C sig!the (?match D)"
      by (simp add: wt_instr_def eff_def xcpt_eff_def)
    with in_any match' obtain
      pc: "handler_pc < length ins"
      "G ⊢ Some ([Class D], LT) <=' phi C sig ! handler_pc"
      by auto
    then obtain ST' LT' where
      phi': "phi C sig ! handler_pc = Some (ST',LT')" and
      less: "G ⊢ ([Class D], LT) <=s (ST',LT')"
      by auto

    from cr' phi meth f'
    have "correct_frame G hp (ST, LT) maxl ins f'"
```

```
      by (unfold correct_state_def) auto
    then obtain
     len: "length loc = 1+length (snd sig)+maxl" and
     loc: "approx_loc G hp loc LT"
      by (unfold correct_frame_def) auto

    let ?f = "([xcp], loc, C, sig, handler_pc)"
    have "correct_frame G hp (ST', LT') maxl ins ?f"
    proof -
      from wf less loc
      have "approx_loc G hp loc LT'" by (simp add: sup_state_conv) blast
      moreover
      from D adr hp
      have "G,hp ⊢ xcp ::≼ Class D" by (simp add: conf_def obj_ty_def)
      with wf less loc
      have "approx_stk G hp [xcp] ST'"
        by (auto simp add: sup_state_conv approx_stk_def approx_val_def
                  elim: conf_widen split: Err.split)
      moreover
      note len pc
      ultimately
      show ?thesis by (simp add: correct_frame_def)
    qed

    with cr' match phi' meth
    show ?thesis by (unfold correct_state_def) auto
  qed
qed

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 -
  note [simp] = split_beta raise_system_xcpt_def
  note [split] = split_if_asm option.split_asm

  assume wt: ?wt ?correct
  hence pre: "preallocated hp" by (simp add: correct_state_def)

  assume xcpt: ?xcpt with pre show ?thesis
  proof (cases "ins!pc")
```

```
    case New with xcpt pre
    show ?thesis by (auto dest: new_Addr_OutOfMemory dest!: preallocatedD)
  next
    case Throw with xcpt wt
    show ?thesis
      by (auto simp add: wt_instr_def correct_state_def correct_frame_def
               dest: non_npD dest!: preallocatedD)
  qed (auto dest!: preallocatedD)
qed


lemma cname_of_xcp [intro]:
  "⟦preallocated hp; xcp = Addr (XcptRef x)⟧ ⟹ cname_of hp xcp = Xcpt x"
  by (auto elim: preallocatedE [of hp x])
```

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 -
  assume wtp: "wt_jvm_prog G phi"
  assume meth: "method (G,C) sig = Some (C,rT,maxs,maxl,ins,et)"
  assume wt: "wt_instr (ins!pc) G rT (phi C sig) maxs (length ins) et pc"
  assume xp: "fst (exec_instr (ins!pc) G hp stk loc C sig pc frs) = Some xcp"
  assume s': "Some state' = exec (G, None, hp, (stk,loc,C,sig,pc)#frs)"
  assume correct: "G,phi ⊢JVM (None, hp, (stk,loc,C,sig,pc)#frs)√"

  from wtp obtain wfmb where wf: "wf_prog wfmb G" by (simp add: wt_jvm_prog_def)

  note xp' = meth s' xp

  note wtp
  moreover
  from xp wt correct
  obtain adr T where
    adr: "xcp = Addr adr" "hp adr = Some T"
    by (blast dest: exec_instr_xcpt_hp)
  moreover
  note correct
  ultimately
  have "G,phi ⊢JVM find_handler G (Some xcp) hp frs √" by (rule uncaught_xcpt_correct)
  with xp'
  have "match_exception_table G (cname_of hp xcp) pc et = None ⟹ ?thesis"
    (is "?m (cname_of hp xcp) = _ ⟹ _" is "?match = _ ⟹ _")
    by (clarsimp simp add: exec_instr_xcpt split_beta)
  moreover
```

```
{ fix handler
  assume some_handler: "?match = Some handler"

  from correct meth
  obtain ST LT where
    hp_ok:  "G ⊢h hp √" and
    prehp:  "preallocated hp" and
    class:  "is_class G C" and
    phi_pc: "phi C sig ! pc = Some (ST, LT)" and
    frame:  "correct_frame G hp (ST, LT) maxl ins (stk, loc, C, sig, pc)" and
    frames: "correct_frames G hp phi rT sig frs"
    by (unfold correct_state_def) auto

  from frame obtain
    stk: "approx_stk G hp stk ST" and
    loc: "approx_loc G hp loc LT" and
    pc:  "pc < length ins" and
    len: "length loc = 1+length (snd sig)+maxl"
    by (unfold correct_frame_def) auto

  from wt obtain
    eff: "∀(pc', s')∈set (xcpt_eff (ins!pc) G pc (phi C sig!pc) et).
          pc' < length ins ∧ G ⊢ s' <=' phi C sig!pc'"
    by (simp add: wt_instr_def eff_def)

  from some_handler xp'
  have state':
    "state' = (None, hp, ([xcp], loc, C, sig, handler)#frs)"
    by (cases "ins!pc", auto simp add: raise_system_xcpt_def split_beta
                              split: split_if_asm)

  let ?f' = "([xcp], loc, C, sig, handler)"
  from eff
  obtain ST' LT' where
    phi_pc': "phi C sig ! handler = Some (ST', LT')" and
    frame': "correct_frame G hp (ST',LT') maxl ins ?f'"
  proof (cases "ins!pc")
    case Return — can't generate exceptions:
    with xp' have False by (simp add: split_beta split: split_if_asm)
    thus ?thesis ..
  next
    case New
    with some_handler xp'
    have xcp: "xcp = Addr (XcptRef OutOfMemory)"
      by (simp add: raise_system_xcpt_def split_beta new_Addr_OutOfMemory)
    with prehp have "cname_of hp xcp = Xcpt OutOfMemory" ..
    with New some_handler phi_pc eff
    obtain ST' LT' where
      phi': "phi C sig ! handler = Some (ST', LT')" and
      less: "G ⊢ ([Class (Xcpt OutOfMemory)], LT) <=s (ST', LT')" and
      pc':  "handler < length ins"
      by (simp add: xcpt_eff_def match_et_imp_match) blast
    note phi'
    moreover
```

```
    { from xcp prehp
      have "G,hp ⊢ xcp ::≼ Class (Xcpt OutOfMemory)"
        by (auto simp add: conf_def obj_ty_def dest!: preallocatedD)
      moreover
      from wf less loc
      have "approx_loc G hp loc LT'"
        by (simp add: sup_state_conv) blast
      moreover
      note wf less pc' len
      ultimately
      have "correct_frame G hp (ST',LT') maxl ins ?f'"
        by (unfold correct_frame_def) (auto simp add: sup_state_conv
            approx_stk_def approx_val_def split: err.split elim: conf_widen)
    }
    ultimately
    show ?thesis by (rule that)
  next
    case Getfield
    with some_handler xp'
    have xcp: "xcp = Addr (XcptRef NullPointer)"
      by (simp add: raise_system_xcpt_def split_beta split: split_if_asm)
    with prehp have "cname_of hp xcp = Xcpt NullPointer" ..
    with Getfield some_handler phi_pc eff
    obtain ST' LT' where
      phi': "phi C sig ! handler = Some (ST', LT')" and
      less: "G ⊢ ([Class (Xcpt NullPointer)], LT) <=s (ST', LT')" and
      pc':  "handler < length ins"
      by (simp add: xcpt_eff_def match_et_imp_match) blast
    note phi'
    moreover
    { from xcp prehp
      have "G,hp ⊢ xcp ::≼ Class (Xcpt NullPointer)"
        by (auto simp add: conf_def obj_ty_def dest!: preallocatedD)
      moreover
      from wf less loc
      have "approx_loc G hp loc LT'"
        by (simp add: sup_state_conv) blast
      moreover
      note wf less pc' len
      ultimately
      have "correct_frame G hp (ST',LT') maxl ins ?f'"
        by (unfold correct_frame_def) (auto simp add: sup_state_conv
            approx_stk_def approx_val_def split: err.split elim: conf_widen)
    }
    ultimately
    show ?thesis by (rule that)
  next
    case Putfield
    with some_handler xp'
    have xcp: "xcp = Addr (XcptRef NullPointer)"
      by (simp add: raise_system_xcpt_def split_beta split: split_if_asm)
    with prehp have "cname_of hp xcp = Xcpt NullPointer" ..
    with Putfield some_handler phi_pc eff
    obtain ST' LT' where
```

```
    phi': "phi C sig ! handler = Some (ST', LT')" and
    less: "G ⊢ ([Class (Xcpt NullPointer)], LT) <=s (ST', LT')" and
    pc':  "handler < length ins"
    by (simp add: xcpt_eff_def match_et_imp_match) blast
  note phi'
  moreover
  { from xcp prehp
    have "G,hp ⊢ xcp ::≼ Class (Xcpt NullPointer)"
      by (auto simp add: conf_def obj_ty_def dest!: preallocatedD)
    moreover
    from wf less loc
    have "approx_loc G hp loc LT'"
      by (simp add: sup_state_conv) blast
    moreover
    note wf less pc' len
    ultimately
    have "correct_frame G hp (ST',LT') maxl ins ?f'"
      by (unfold correct_frame_def) (auto simp add: sup_state_conv
          approx_stk_def approx_val_def split: err.split elim: conf_widen)
  }
  ultimately
  show ?thesis by (rule that)
next
  case Checkcast
  with some_handler xp'
  have xcp: "xcp = Addr (XcptRef ClassCast)"
    by (simp add: raise_system_xcpt_def split_beta split: split_if_asm)
  with prehp have "cname_of hp xcp = Xcpt ClassCast" ..
  with Checkcast some_handler phi_pc eff
  obtain ST' LT' where
    phi': "phi C sig ! handler = Some (ST', LT')" and
    less: "G ⊢ ([Class (Xcpt ClassCast)], LT) <=s (ST', LT')" and
    pc':  "handler < length ins"
    by (simp add: xcpt_eff_def match_et_imp_match) blast
  note phi'
  moreover
  { from xcp prehp
    have "G,hp ⊢ xcp ::≼ Class (Xcpt ClassCast)"
      by (auto simp add: conf_def obj_ty_def dest!: preallocatedD)
    moreover
    from wf less loc
    have "approx_loc G hp loc LT'"
      by (simp add: sup_state_conv) blast
    moreover
    note wf less pc' len
    ultimately
    have "correct_frame G hp (ST',LT') maxl ins ?f'"
      by (unfold correct_frame_def) (auto simp add: sup_state_conv
          approx_stk_def approx_val_def split: err.split elim: conf_widen)
  }
  ultimately
  show ?thesis by (rule that)
next
  case Invoke
```

```
    with phi_pc eff
    have
      "∀D∈set (match_any G pc et).
      the (?m D) < length ins ∧ G ⊢ Some ([Class D], LT) <=' phi C sig!the (?m D)"
      by (simp add: xcpt_eff_def)
    moreover
    from some_handler
    obtain D where
      "D ∈ set (match_any G pc et)" and
      D: "G ⊢ cname_of hp xcp ⪯C D" and
      "?m D = Some handler"
      by (blast dest: in_match_any)
    ultimately
    obtain
      pc': "handler < length ins" and
      "G ⊢ Some ([Class D], LT) <=' phi C sig ! handler"
      by auto
    then
    obtain ST' LT' where
      phi': "phi C sig ! handler = Some (ST', LT')" and
      less: "G ⊢ ([Class D], LT) <=s (ST', LT')"
      by auto
    from xp wt correct
    obtain addr T where
      xcp: "xcp = Addr addr" "hp addr = Some T"
      by (blast dest: exec_instr_xcpt_hp)
    note phi'
    moreover
    { from xcp D
      have "G,hp ⊢ xcp ::≼ Class D"
        by (simp add: conf_def obj_ty_def)
      moreover
      from wf less loc
      have "approx_loc G hp loc LT'"
        by (simp add: sup_state_conv) blast
      moreover
      note wf less pc' len
      ultimately
      have "correct_frame G hp (ST',LT') maxl ins ?f'"
        by (unfold correct_frame_def) (auto simp add: sup_state_conv
            approx_stk_def approx_val_def split: err.split elim: conf_widen)
    }
    ultimately
    show ?thesis by (rule that)
  next
    case Throw
    with phi_pc eff
    have
      "∀D∈set (match_any G pc et).
      the (?m D) < length ins ∧ G ⊢ Some ([Class D], LT) <=' phi C sig!the (?m D)"
      by (simp add: xcpt_eff_def)
    moreover
    from some_handler
    obtain D where
```

```
      "D ∈ set (match_any G pc et)" and
      D: "G ⊢ cname_of hp xcp ⪯C D" and
      "?m D = Some handler"
      by (blast dest: in_match_any)
    ultimately
    obtain
      pc': "handler < length ins" and
      "G ⊢ Some ([Class D], LT) <=' phi C sig ! handler"
      by auto
    then
    obtain ST' LT' where
      phi': "phi C sig ! handler = Some (ST', LT')" and
      less: "G ⊢ ([Class D], LT) <=s (ST', LT')"
      by auto
    from xp wt correct
    obtain addr T where
      xcp: "xcp = Addr addr" "hp addr = Some T"
      by (blast dest: exec_instr_xcpt_hp)
    note phi'
    moreover
    { from xcp D
      have "G,hp ⊢ xcp ::⪯ Class D"
        by (simp add: conf_def obj_ty_def)
      moreover
      from wf less loc
      have "approx_loc G hp loc LT'"
        by (simp add: sup_state_conv) blast
      moreover
      note wf less pc' len
      ultimately
      have "correct_frame G hp (ST',LT') maxl ins ?f'"
        by (unfold correct_frame_def) (auto simp add: sup_state_conv
            approx_stk_def approx_val_def split: err.split elim: conf_widen)
    }
    ultimately
    show ?thesis by (rule that)
  qed (insert xp', auto) — the other instructions don't generate exceptions

  from state' meth hp_ok class frames phi_pc' frame'
  have ?thesis by (unfold correct_state_def) simp
 }
 ultimately
 show ?thesis by (cases "?match") blast+
qed
```

4.22.3 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'√"
apply (clarsimp simp add: defs1 map_eq_Cons)
apply (blast intro: approx_loc_imp_approx_val_sup)
done

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'√"
apply (clarsimp simp add: defs1 map_eq_Cons)
apply (blast intro: approx_loc_subst)
done


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'√"
apply (clarsimp simp add: defs1 sup_PTS_eq map_eq_Cons)
apply (blast dest: conf_litval intro: conf_widen)
done


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"
apply (unfold cast_ok_def)
apply (frule widen_Class)
apply (elim exE disjE)
 apply (simp add: null)
apply (clarsimp simp add: conf_def obj_ty_def)
apply (cases v)
apply (auto intro: rtrancl_trans)
```

done

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'√*"
apply *(clarsimp simp add: defs2 wt_jvm_prog_def map_eq_Cons split: split_if_asm)*
apply *(blast intro: Cast_conf2)*
done

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'√*"
apply *(clarsimp simp add: defs2 map_eq_Cons wt_jvm_prog_def split_beta*
split: option.split split_if_asm)
apply *(frule conf_widen)*
apply *assumption+*
apply *(drule conf_RefTD)*
apply *(clarsimp simp add: defs2)*
apply *(rule conjI)*
apply *(drule widen_cfs_fields)*
apply *assumption+*
apply *(erule conf_widen)*
prefer *2*
apply *assumption*
apply *(simp add: hconf_def oconf_def lconf_def)*
apply *(elim allE)*
apply *(erule impE, assumption)*
apply *simp*
apply *(elim allE)*
apply *(erule impE, assumption)*
apply *clarsimp*
apply *blast*
done

```
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'√"
apply (clarsimp simp add: defs2 split_beta split: option.split List.split split_if_asm)
apply (frule conf_widen)
prefer 2
  apply assumption
 apply assumption
apply (drule conf_RefTD)
apply clarsimp
apply (blast
       intro:
         hext_upd_obj approx_stk_sup_heap
         approx_loc_sup_heap
         hconf_field_update
         preallocated_field_update
         correct_frames_field_update conf_widen
       dest:
         widen_cfs_fields)
done


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 -
  assume wf:   "wf_prog wt G"
  assume meth: "method (G,C) sig = Some (C,rT,maxs,maxl,ins,et)"
  assume ins:  "ins!pc = New X"
  assume wt:   "wt_instr (ins!pc) G rT (phi C sig) maxs (length ins) et pc"
  assume exec: "Some state' = exec (G, None, hp, (stk,loc,C,sig,pc)#frs)"
  assume conf: "G,phi ⊢JVM (None, hp, (stk,loc,C,sig,pc)#frs)√"
  assume no_x: "fst (exec_instr (ins!pc) G hp stk loc C sig pc frs) = None"
```

```
from ins conf meth
obtain ST LT where
  heap_ok:    "G⊢h hp√" and
  prealloc:   "preallocated hp" and
  phi_pc:     "phi C sig!pc = Some (ST,LT)" and
  is_class_C: "is_class G C" and
  frame:      "correct_frame G hp (ST,LT) maxl ins (stk, loc, C, sig, pc)" and
  frames:     "correct_frames G hp phi rT sig frs"
  by (auto simp add: correct_state_def iff del: not_None_eq)

from phi_pc ins wt
obtain ST' LT' where
  is_class_X: "is_class G X" and
  maxs:       "length ST < maxs" and
  suc_pc:     "Suc pc < length ins" and
  phi_suc:    "phi C sig ! Suc pc = Some (ST', LT')" and
  less:       "G ⊢ (Class X # ST, LT) <=s (ST', LT')"
  by (unfold wt_instr_def eff_def norm_eff_def) auto

obtain oref xp' where
  new_Addr: "new_Addr hp = (oref,xp')"
  by (cases "new_Addr hp")
with ins no_x
obtain hp: "hp oref = None" and "xp' = None"
  by (auto dest: new_AddrD simp add: raise_system_xcpt_def)

with exec ins meth new_Addr
have state':
  "state' = Norm (hp(oref↦(X, init_vars (fields (G, X)))),
            (Addr oref # stk, loc, C, sig, Suc pc) # frs)"
  (is "state' = Norm (?hp', ?f # frs)")
  by simp
moreover
from wf hp heap_ok is_class_X
have hp': "G ⊢h ?hp' √"
  by - (rule hconf_newref, auto simp add: oconf_def dest: fields_is_type)
moreover
from hp
have sup: "hp ≤| ?hp'" by (rule hext_new)
from hp frame less suc_pc wf
have "correct_frame G ?hp' (ST', LT') maxl ins ?f"
  apply (unfold correct_frame_def sup_state_conv)
  apply (clarsimp simp add: map_eq_Cons conf_def fun_upd_apply approx_val_def)
  apply (blast intro: approx_stk_sup_heap approx_loc_sup_heap sup)
  done
moreover
from hp frames wf heap_ok is_class_X
have "correct_frames G ?hp' phi rT sig frs"
```

```
    by - (rule correct_frames_newref,
          auto simp add: oconf_def dest: fields_is_type)
  moreover
  from hp prealloc have "preallocated ?hp'" by (rule preallocated_newref)
  ultimately
  show ?thesis
    by (simp add: is_class_C meth phi_suc correct_state_def del: not_None_eq)
qed

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 -
  assume wtprog: "wt_jvm_prog G phi"
  assume method: "method (G,C) sig = Some (C,rT,maxs,maxl,ins,et)"
  assume ins:    "ins ! pc = Invoke C' mn pTs"
  assume wti:    "wt_instr (ins!pc) G rT (phi C sig) maxs (length ins) et pc"
  assume state': "Some state' = exec (G, None, hp, (stk,loc,C,sig,pc)#frs)"
  assume approx: "G,phi ⊢JVM (None, hp, (stk,loc,C,sig,pc)#frs)√"
  assume no_xcp: "fst (exec_instr (ins!pc) G hp stk loc C sig pc frs) = None"

  from wtprog
  obtain wfmb where
    wfprog: "wf_prog wfmb G"
    by (simp add: wt_jvm_prog_def)

  from ins method approx
  obtain s where
    heap_ok: "G⊢h hp√" and
    prealloc:"preallocated hp" and
    phi_pc:  "phi C sig!pc = Some s" and
    is_class_C: "is_class G C" and
    frame:   "correct_frame G hp s maxl ins (stk, loc, C, sig, pc)" and
    frames:  "correct_frames G hp phi rT sig frs"
    by (clarsimp simp add: correct_state_def iff del: not_None_eq)

  from ins wti phi_pc
  obtain apTs X ST LT D' rT body where
    is_class: "is_class G C'" and
```

```
  s:  "s = (rev apTs @ X # ST, LT)" and
  l:  "length apTs = length pTs" and
  X:  "G⊢ X ≼ Class C'" and
  w:  "∀x∈set (zip apTs pTs). x ∈ widen G" and
  mC':"method (G, C') (mn, pTs) = Some (D', rT, body)" and
  pc: "Suc pc < length ins" and
  eff: "G ⊢ norm_eff (Invoke C' mn pTs) G (Some s) <=' phi C sig!Suc pc"
  by (simp add: wt_instr_def eff_def del: not_None_eq)
     (elim exE conjE, rule that)

from eff
obtain ST' LT' where
  s': "phi C sig ! Suc pc = Some (ST', LT')"
  by (simp add: norm_eff_def split_paired_Ex) blast

from X
obtain T where
  X_Ref: "X = RefT T"
  by - (drule widen_RefT2, erule exE, rule that)

from s ins frame
obtain
  a_stk: "approx_stk G hp stk (rev apTs @ X # ST)" and
  a_loc: "approx_loc G hp loc LT" and
  suc_l: "length loc = Suc (length (snd sig) + maxl)"
  by (simp add: correct_frame_def)

from a_stk
obtain opTs stk' oX where
  opTs:   "approx_stk G hp opTs (rev apTs)" and
  oX:     "approx_val G hp oX (OK X)" and
  a_stk': "approx_stk G hp stk' ST" and
  stk':   "stk = opTs @ oX # stk'" and
  l_o:    "length opTs = length apTs"
          "length stk' = length ST"
  by - (drule approx_stk_append, auto)

from oX X_Ref
have oX_conf: "G,hp ⊢ oX ::≼ RefT T"
  by (simp add: approx_val_def)

from stk' l_o l
have oX_pos: "last (take (Suc (length pTs)) stk) = oX" by simp

with state' method ins no_xcp oX_conf
obtain ref where oX_Addr: "oX = Addr ref"
  by (auto simp add: raise_system_xcpt_def dest: conf_RefTD)
```

```
  with oX_conf X_Ref
  obtain obj D where
    loc:    "hp ref = Some obj" and
    obj_ty: "obj_ty obj = Class D" and
    D:      "G ⊢ Class D ≼ X"
    by (auto simp add: conf_def) blast

  with X_Ref obtain X' where X': "X = Class X'"
    by (blast dest: widen_Class)

  with X have X'_subcls: "G ⊢ X' ≼C C'"  by simp

  with mC' wfprog
  obtain D0 rT0 maxs0 maxl0 ins0 et0 where
    mX: "method (G, X') (mn, pTs) = Some (D0, rT0, maxs0, maxl0, ins0, et0)" "G⊢rT0≼rT"
    by (auto dest: subtype_widen_methd intro: that)

  from X' D have D_subcls: "G ⊢ D ≼C X'" by simp

  with wfprog mX
  obtain D'' rT' mxs' mxl' ins' et' where
    mD: "method (G, D) (mn, pTs) = Some (D'', rT', mxs', mxl', ins', et')"
        "G ⊢ rT' ≼ rT0"
    by (auto dest: subtype_widen_methd intro: that)

  from mX mD have rT': "G ⊢ rT' ≼ rT" by - (rule widen_trans)

  from is_class X'_subcls D_subcls
  have is_class_D: "is_class G D" by (auto dest: subcls_is_class2)

  with mD wfprog
  obtain mD'':
    "method (G, D'') (mn, pTs) = Some (D'', rT', mxs', mxl', ins', et')"
    "is_class G D''"
    by (auto dest: method_in_md)

  from loc obj_ty have "fst (the (hp ref)) = D" by (simp add: obj_ty_def)

  with oX_Addr oX_pos state' method ins stk' l_o l loc obj_ty mD no_xcp
  have state'_val:
    "state' =
     Norm (hp, ([], Addr ref # rev opTs @ replicate mxl' arbitrary,
                D'', (mn, pTs), 0) # (opTs @ Addr ref # stk', loc, C, sig, pc) # frs)"
    (is "state' = Norm (hp, ?f # ?f' # frs)")
    by (simp add: raise_system_xcpt_def)

  from wtprog mD''
  have start: "wt_start G D'' pTs mxl' (phi D'' (mn, pTs)) ∧ ins' ≠ []"
```

```
    by (auto dest: wt_jvm_prog_impl_wt_start)

  then obtain LT0 where
    LT0: "phi D'' (mn, pTs) ! 0 = Some ([], LT0)"
    by (clarsimp simp add: wt_start_def sup_state_conv)

  have c_f: "correct_frame G hp ([], LT0) mxl' ins' ?f"
  proof -
    from start LT0
    have sup_loc:
      "G ⊢ (OK (Class D'')) # map OK pTs @ replicate mxl' Err) <=l LT0"
      (is "G ⊢ ?LT <=l LT0")
      by (simp add: wt_start_def sup_state_conv)

    have r: "approx_loc G hp (replicate mxl' arbitrary) (replicate mxl' Err)"
      by (simp add: approx_loc_def list_all2_def set_replicate_conv_if)

    from wfprog mD is_class_D
    have "G ⊢ Class D ≼ Class D''"
      by (auto dest: method_wf_mdecl)
    with obj_ty loc
    have a: "approx_val G hp (Addr ref) (OK (Class D''))"
      by (simp add: approx_val_def conf_def)

    from opTs w l l_o wfprog
    have "approx_stk G hp opTs (rev pTs)"
      by (auto elim!: approx_stk_all_widen simp add: zip_rev)
    hence "approx_stk G hp (rev opTs) pTs" by (subst approx_stk_rev)

    with r a l_o l
    have "approx_loc G hp (Addr ref # rev opTs @ replicate mxl' arbitrary) ?LT"
      (is "approx_loc G hp ?lt ?LT")
      by (auto simp add: approx_loc_append approx_stk_def)

    from this sup_loc wfprog
    have "approx_loc G hp ?lt LT0" by (rule approx_loc_widen)
    with start l_o l
    show ?thesis by (simp add: correct_frame_def)
  qed

  from state'_val heap_ok mD'' ins method phi_pc s X' l mX
       frames s' LT0 c_f is_class_C stk' oX_Addr frame prealloc
  show ?thesis by (simp add: correct_state_def) (intro exI conjI, blast)
qed

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 -
  assume wt_prog: "wt_jvm_prog G phi"
  assume meth: "method (G,C) sig = Some (C,rT,maxs,maxl,ins,et)"
  assume ins: "ins ! pc = Return"
  assume wt: "wt_instr (ins!pc) G rT (phi C sig) maxs (length ins) et pc"
  assume s': "Some state' = exec (G, None, hp, (stk,loc,C,sig,pc)#frs)"
  assume correct: "G,phi ⊢JVM (None, hp, (stk,loc,C,sig,pc)#frs)√"

  from wt_prog
  obtain wfmb where wf: "wf_prog wfmb G" by (simp add: wt_jvm_prog_def)

  from meth ins s'
  have "frs = [] ⟹ ?thesis" by (simp add: correct_state_def)
  moreover
  { fix f frs'
    assume frs': "frs = f#frs'"
    moreover
    obtain stk' loc' C' sig' pc' where
      f: "f = (stk',loc',C',sig',pc')" by (cases f)
    moreover
    obtain mn pt where
      sig: "sig = (mn,pt)" by (cases sig)
    moreover
    note meth ins s'
    ultimately
    have state':
      "state' = (None,hp,(hd stk#(drop (1+length pt) stk'),loc',C',sig',pc'+1)#frs')"
      (is "state' = (None,hp,?f'#frs')")
      by simp

    from correct meth
    obtain ST LT where
      hp_ok:  "G ⊢h hp √" and
      alloc:  "preallocated hp" and
      phi_pc: "phi C sig ! pc = Some (ST, LT)" and
      frame:  "correct_frame G hp (ST, LT) maxl ins (stk,loc,C,sig,pc)" and
      frames: "correct_frames G hp phi rT sig frs"
      by (simp add: correct_state_def, clarify, blast)

    from phi_pc ins wt
    obtain T ST' where "ST = T # ST'" "G ⊢ T ⪯ rT"
```

```
  by (simp add: wt_instr_def) blast
with wf frame
have hd_stk: "G,hp ⊢ (hd stk) ::≼ rT"
  by (auto simp add: correct_frame_def elim: conf_widen)

from f frs' frames sig
obtain apTs ST0' ST' LT' D D' D'' rT' rT'' maxs' maxl' ins' et' body where
  phi':   "phi C' sig' ! pc' = Some (ST',LT')" and
  class': "is_class G C'" and
  meth':  "method (G,C') sig' = Some (C',rT',maxs',maxl',ins',et')" and
  ins':   "ins' ! pc' = Invoke D' mn pt" and
  frame': "correct_frame G hp (ST', LT') maxl' ins' f" and
  frames':"correct_frames G hp phi rT' sig' frs'" and
  rT'':   "G ⊢ rT ≼ rT''" and
  meth'': "method (G, D) sig = Some (D'', rT'', body)" and
  ST0':   "ST' = rev apTs @ Class D # ST0'" and
  len':   "length apTs = length pt"
  by clarsimp blast

from f frame'
obtain
  stk': "approx_stk G hp stk' ST'" and
  loc': "approx_loc G hp loc' LT'" and
  pc':  "pc' < length ins'" and
  lloc':"length loc' = Suc (length (snd sig') + maxl')"
  by (simp add: correct_frame_def)

from wt_prog class' meth' pc'
have "wt_instr (ins'!pc') G rT' (phi C' sig') maxs' (length ins') et' pc'"
  by (rule wt_jvm_prog_impl_wt_instr)
with ins' phi' sig
obtain apTs ST0 X ST'' LT'' body' rT0 mD where
  phi_suc: "phi C' sig' ! Suc pc' = Some (ST'', LT'')" and
  ST0:     "ST' = rev apTs @ X # ST0" and
  len:     "length apTs = length pt" and
  less:    "G ⊢ (rT0 # ST0, LT') <=s (ST'', LT'')" and
  methD':  "method (G, D') sig = Some (mD, rT0, body')" and
  lessD':  "G ⊢ X ≼ Class D'" and
  suc_pc': "Suc pc' < length ins'"
  by (clarsimp simp add: wt_instr_def eff_def norm_eff_def) blast

from len len' ST0 ST0'
have "X = Class D" by simp
with lessD'
have "G ⊢ D ≼C D'" by simp
moreover
note wf meth'' methD'
ultimately
```

```
    have "G ⊢ rT'' ≼ rT0" by (auto dest: subcls_widen_methd)
    with wf hd_stk rT''
    have hd_stk': "G,hp ⊢ (hd stk) ::≼ rT0" by (auto elim: conf_widen widen_trans)

    have frame'':
      "correct_frame G hp (ST'',LT'') maxl' ins' ?f'"
    proof -
      from wf hd_stk' len stk' less ST0
      have "approx_stk G hp (hd stk # drop (1+length pt) stk') ST''"
        by (auto simp add: map_eq_Cons sup_state_conv
                  dest!: approx_stk_append elim: conf_widen)
      moreover
      from wf loc' less
      have "approx_loc G hp loc' LT''" by (simp add: sup_state_conv) blast
      moreover
      note suc_pc' lloc'
      ultimately
      show ?thesis by (simp add: correct_frame_def)
    qed

    with state' frs' f meth hp_ok hd_stk phi_suc frames' meth' phi' class' alloc
    have ?thesis by (simp add: correct_state_def)
  }
  ultimately
  show ?thesis by (cases frs) blast+
qed

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'√"
apply (clarsimp simp add: defs2)
apply fast
done


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'√"
apply (clarsimp simp add: defs2)
apply fast
done

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'√"
apply (clarsimp simp add: defs2)
apply fast
done

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'√"
apply (clarsimp simp add: defs2 map_eq_Cons)
apply (blast intro: conf_widen)
done

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'√"
apply (clarsimp simp add: defs2 map_eq_Cons)
apply (blast intro: conf_widen)
done

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'√"
apply (clarsimp simp add: defs2 map_eq_Cons)
apply (blast intro: conf_widen)
done

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'√"
apply (clarsimp simp add: defs2 map_eq_Cons)
apply (blast intro: conf_widen)
done

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'√"
apply (clarsimp simp add: defs2 map_eq_Cons approx_val_def conf_def)
apply blast
done

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'√"
  by simp
```

The next theorem collects the results of the sections above, i.e. exception handling and the execution step for each instruction. It states type safety for single step execution: in 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'√"
apply (frule wt_jvm_prog_impl_wt_instr_cor)
apply assumption+
apply (cases "fst (exec_instr (ins!pc) G hp stk loc C sig pc frs)")
defer
apply (erule xcpt_correct, assumption+)
apply (cases "ins!pc")
prefer 8
apply (rule Invoke_correct, assumption+)
prefer 8
apply (rule Return_correct, assumption+)
prefer 5
apply (rule Getfield_correct, assumption+)
prefer 6
apply (rule Checkcast_correct, assumption+)

apply (unfold wt_jvm_prog_def)
apply (rule Load_correct, assumption+)
apply (rule Store_correct, assumption+)
apply (rule LitPush_correct, assumption+)
apply (rule New_correct, assumption+)
apply (rule Putfield_correct, assumption+)
apply (rule Pop_correct, assumption+)
apply (rule Dup_correct, assumption+)
apply (rule Dup_x1_correct, assumption+)
apply (rule Dup_x2_correct, assumption+)
apply (rule Swap_correct, assumption+)
apply (rule IAdd_correct, assumption+)
apply (rule Goto_correct, assumption+)
apply (rule Ifcmpeq_correct, assumption+)
apply (rule Throw_correct, assumption+)
done
```

4.22.4 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)"
by (auto simp add: correct_state_def Let_def)


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'√"
apply (simp only: split_tupled_all)
apply (rename_tac xp hp frs)
apply (case_tac xp)
 apply (case_tac frs)
  apply simp
 apply (simp only: split_tupled_all)
 apply hypsubst
 apply (frule correct_state_impl_Some_method)
 apply (force intro: instr_correct)
apply (case_tac frs)
```

```
apply simp_all
done



lemma L0:
  "⟦ xp=None; frs≠[] ⟧ ⟹ (∃state'. exec (G,xp,hp,frs) = Some state')"
by (clarsimp simp add: neq_Nil_conv split_beta)

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'√"
apply (drule L0)
apply assumption
apply (fast intro: BV_correct_1)
done

theorem BV_correct [rule_format]:
"⟦ wt_jvm_prog G phi; G ⊢ s -jvm→ t ⟧ ⟹ G,phi ⊢JVM s√ ⟶ G,phi ⊢JVM t√"
apply (unfold exec_all_def)
apply (erule rtrancl_induct)
 apply simp
apply (auto intro: BV_correct_1)
done



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)))"
apply (drule BV_correct)
apply assumption+
apply (simp add: correct_state_def correct_frame_def split_def
            split: option.splits)
done

lemma
  fixes G :: jvm_prog ("Γ")
  assumes wf: "wf_prog wf_mb Γ"
  shows hconf_start: "Γ ⊢h (start_heap Γ) √"
  apply (unfold hconf_def start_heap_def)
  apply (auto simp add: fun_upd_apply blank_def oconf_def split: split_if_asm)
  apply (simp add: fields_is_type [OF _ wf is_class_xcpt [OF wf]])+
  done

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 √"
  apply (cases b)
  apply (unfold  start_state_def)
  apply (unfold correct_state_def)
```

apply *(auto simp add: preallocated_start)*
apply *(simp add: wt_jvm_prog_def hconf_start)*
apply *(drule wt_jvm_prog_impl_wt_start, assumption+)*
apply *(clarsimp simp add: wt_start_def)*
apply *(auto simp add: correct_frame_def)*
apply *(simp add: approx_stk_def sup_state_conv)*
apply *(auto simp add: sup_state_conv approx_val_def dest!: widen_RefT split: err.splits)*
done

theorem
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)"
shows *typesafe:*
"G ⊢ start_state Γ C m -jvm→ s ⟹ Γ,Φ ⊢JVM s √"
proof -
from *welltyped main_method*
have *"Γ,Φ ⊢JVM start_state Γ C m √"* **by** *(rule BV_correct_initial)*
moreover
assume *"G ⊢ start_state Γ C m -jvm→ s"*
ultimately
show *"Γ,Φ ⊢JVM s √"* **using** *welltyped* **by** - *(rule BV_correct)*
qed

end

4.23 Welltyped Programs produce no Type Errors

theory *BVNoTypeError = JVMDefensive + BVSpecTypeSafe:*

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"
by *(cases v) auto*

lemma *isRef_def2:*
"isRef v = (v = Null ∨ (∃loc. v = Addr loc))"
by *(cases v) (auto simp add: isRef_def)*

lemma *list_all2I:*
"∀x ∈ set (zip a b). split P x ⟹ length a = length b ⟹ list_all2 P a b"
by *(simp add: list_all2_def)*

lemma *app'Store[simp]:*
"app' (Store idx, G, pc, maxs, rT, (ST,LT)) = (∃T ST'. ST = T#ST' ∧ idx < length LT)"
by *(cases ST, auto)*

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)"
by *(cases ST, auto)*

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')"
apply *rule*
defer
apply *clarsimp*
apply *(cases ST)*
apply *simp*
apply *(cases "tl ST")*
apply *auto*
done

lemma *app'Checkcast[simp]:*
"app' (Checkcast C, G, pc, maxs, rT, (ST,LT)) =
(∃rT ST'. ST = RefT rT#ST' ∧ is_class G C)"
apply *rule*

defer
apply *clarsimp*
apply *(cases ST)*
apply *simp*
apply *(cases "hd ST")*
defer
apply *simp*
apply *simp*
done

lemma *app'Pop[simp]:*
"app' (Pop, G, pc, maxs, rT, (ST,LT)) = (∃T ST'. ST = T#ST')"
by *(cases ST, auto)*

lemma *app'Dup[simp]:*
"app' (Dup, G, pc, maxs, rT, (ST,LT)) =
(∃T ST'. ST = T#ST' ∧ length ST < maxs)"
by *(cases ST, auto)*

lemma *app'Dup_x1[simp]:*
"app' (Dup_x1, G, pc, maxs, rT, (ST,LT)) =
(∃T1 T2 ST'. ST = T1#T2#ST' ∧ length ST < maxs)"
by *(cases ST, simp, cases "tl ST", auto)*

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)"
by *(cases ST, simp, cases "tl ST", simp, cases "tl (tl ST)", auto)*

lemma *app'Swap[simp]:*
"app' (Swap, G, pc, maxs, rT, (ST,LT)) = (∃T1 T2 ST'. ST = T1#T2#ST')"
by *(cases ST, simp, cases "tl ST", auto)*

lemma *app'IAdd[simp]:*
"app' (IAdd, G, pc, maxs, rT, (ST,LT)) =
(∃ST'. ST = PrimT Integer#PrimT Integer#ST')"
apply *(cases ST)*
apply *simp*
apply *simp*
apply *(case_tac a)*
apply *auto*
apply *(case_tac prim_ty)*

```
  apply auto
  apply (case_tac prim_ty)
  apply auto
  apply (case_tac list)
  apply auto
  apply (case_tac a)
  apply auto
  apply (case_tac prim_ty)
  apply auto
  done


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')))"
  apply auto
  apply (cases ST)
  apply simp
  apply (cases "tl ST")
  apply (case_tac a)
  apply auto
  done


lemma app'Return[simp]:
  "app' (Return, G, pc, maxs, rT, (ST,LT)) =
  (∃T ST'. ST = T#ST'∧ G ⊢ T ≼ rT)"
  by (cases ST, auto)


lemma app'Throw[simp]:
  "app' (Throw, G, pc, maxs, rT, (ST,LT)) =
  (∃ST' r. ST = RefT r#ST')"
  apply (cases ST, simp)
  apply (cases "hd ST")
  apply auto
  done


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
  assume "?P ST LT" thus "?app ST LT" by (auto simp add: min_def list_all2_def)
next
  assume app: "?app ST LT"
  hence l: "length fpTs < length ST" by simp
  obtain xs ys where xs: "ST = xs @ ys" "length xs = length fpTs"
  proof -
    have "ST = take (length fpTs) ST @ drop (length fpTs) ST" by simp
    moreover from l have "length (take (length fpTs) ST) = length fpTs"
      by (simp add: min_def)
    ultimately show ?thesis ..
  qed
  obtain apTs where
    "ST = (rev apTs) @ ys" and "length apTs = length fpTs"
  proof -
    have "ST = rev (rev xs) @ ys" by simp
    with xs show ?thesis by - (rule that, assumption, simp)
  qed
  moreover
  from l xs obtain X ST' where "ys = X#ST'" by (auto simp add: neq_Nil_conv)
  ultimately
  have "ST = (rev apTs) @ X # ST'" "length apTs = length fpTs" by auto
  with app
  show "?P ST LT"
    apply (clarsimp simp add: list_all2_def min_def)
    apply ((rule exI)+, (rule conjI)?)+
    apply auto
    done
qed

lemma approx_loc_len [simp]:
  "approx_loc G hp loc LT ⟹ length loc = length LT"
  by (simp add: approx_loc_def list_all2_def)

lemma approx_stk_len [simp]:
  "approx_stk G hp stk ST ⟹ length stk = length ST"
  by (simp add: approx_stk_def)

lemma isRefI [intro, simp]: "G,hp ⊢ v ::≼ RefT T ⟹ isRef v"
  apply (drule conf_RefTD)
  apply (auto simp add: isRef_def)
  done

lemma isIntgI [intro, simp]: "G,hp ⊢ v ::≼ PrimT Integer ⟹ isIntg v"
  apply (unfold conf_def)
  apply auto
  apply (erule widen.elims)
```

```
  apply auto
  apply (cases v)
  apply auto
  done

lemma list_all2_approx:
  "⋀s. list_all2 (approx_val G hp) s (map OK S) =
       list_all2 (conf G hp) s S"
  apply (induct S)
  apply (auto simp add: list_all2_Cons2 approx_val_def)
  done

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"
  apply (rule list_all2_trans)
  defer
  apply assumption
  apply assumption
  apply (drule conf_widen, assumption+)
  done
```

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 -
  from welltyped obtain mb where wf: "wf_prog mb G" by (fast dest: wt_jvm_progD)

  obtain xcp hp frs where s [simp]: "s = (xcp, hp, frs)" by (cases s)

  from conforms have "xcp ≠ None ∨ frs = [] ⟹ check G s"
    by (unfold correct_state_def check_def) auto
  moreover {
    assume "¬(xcp ≠ None ∨ frs = [])"
    then obtain stk loc C sig pc frs' where
      xcp [simp]: "xcp = None" and
      frs [simp]: "frs = (stk,loc,C,sig,pc)#frs'"
      by (clarsimp simp add: neq_Nil_conv) fast

    from conforms obtain  ST LT rT maxs maxl ins et where
      hconf:  "G ⊢h hp √" and
      class:  "is_class G C" and
      meth:   "method (G, C) sig = Some (C, rT, maxs, maxl, ins, et)" and
      phi:    "Phi C sig ! pc = Some (ST,LT)" and
```

```
  frame:  "correct_frame G hp (ST,LT) maxl ins (stk,loc,C,sig,pc)" and
  frames: "correct_frames G hp Phi rT sig frs'"
  by (auto simp add: correct_state_def)

from frame obtain
  stk: "approx_stk G hp stk ST" and
  loc: "approx_loc G hp loc LT" and
  pc:  "pc < length ins" and
  len: "length loc = length (snd sig) + maxl + 1"
  by (auto simp add: correct_frame_def)

note approx_val_def [simp]

from welltyped meth conforms
have "wt_instr (ins!pc) G rT (Phi C sig) maxs (length ins) et pc"
  by simp (rule wt_jvm_prog_impl_wt_instr_cor)
then obtain
  app': "app (ins!pc) G maxs rT pc et (Phi C sig!pc) " and
  eff: "∀(pc', s')∈set (eff (ins ! pc) G pc et (Phi C sig ! pc)). pc' < length ins"
  by (simp add: wt_instr_def phi) blast

from eff
have pc': "∀pc' ∈ set (succs (ins!pc) pc). pc' < length ins"
  by (simp add: eff_def) blast

from app' phi
have app:
  "xcpt_app (ins!pc) G pc et ∧ app' (ins!pc, G, pc, maxs, rT, (ST,LT))"
  by (clarsimp simp add: app_def)

with eff stk loc pc'
have "check_instr (ins!pc) G hp stk loc C sig pc frs'"
proof (cases "ins!pc")
  case (Getfield F C)
  with app stk loc phi obtain v vT stk' where
    class: "is_class G C" and
    field: "field (G, C) F = Some (C, vT)" and
    stk:   "stk = v # stk'" and
    conf:  "G,hp ⊢ v ::≼ Class C"
    apply clarsimp
    apply (blast dest: conf_widen [OF wf])
    done
  from conf have isRef: "isRef v" ..
  moreover {
    assume "v ≠ Null"
    with conf field isRef wf
    have "∃D vs. hp (the_Addr v) = Some (D,vs) ∧ G ⊢ D ≼C C"
      by (auto dest!: non_np_objD)
```

```
    }
    ultimately show ?thesis using Getfield field class stk hconf
      apply clarsimp
      apply (fastsimp dest!: hconfD widen_cfs_fields [OF _ _ wf] oconf_objD)
      done
  next
    case (Putfield F C)
    with app stk loc phi obtain v ref vT stk' where
      class: "is_class G C" and
      field: "field (G, C) F = Some (C, vT)" and
      stk:   "stk = v # ref # stk'" and
      confv: "G,hp ⊢ v ::≼ vT" and
      confr: "G,hp ⊢ ref ::≼ Class C"
      apply clarsimp
      apply (blast dest: conf_widen [OF wf])
      done
    from confr have isRef: "isRef ref" ..
    moreover {
      assume "ref ≠ Null"
      with confr field isRef wf
      have "∃D vs. hp (the_Addr ref) = Some (D,vs) ∧ G ⊢ D ≼C C"
        by (auto dest: non_np_objD)
    }
    ultimately show ?thesis using Putfield field class stk confv
      by clarsimp
  next
    case (Invoke C mn ps)
    with app
    obtain apTs X ST' where
      ST: "ST = rev apTs @ X # ST'" and
      ps: "length apTs = length ps" and
      w:   "∀x∈set (zip apTs ps). x ∈ widen G" and
      C:   "G ⊢ X ≼ Class C" and
      mth: "method (G, C) (mn, ps) ≠ None"
      by (simp del: app'.simps) blast

    from ST stk
    obtain aps x stk' where
      stk': "stk = aps @ x # stk'" and
      aps: "approx_stk G hp aps (rev apTs)" and
      x: "G,hp ⊢ x ::≼ X" and
      l: "length aps = length apTs"
      by (auto dest!: approx_stk_append)

    from stk' l ps
    have [simp]: "stk!length ps = x" by (simp add: nth_append)
    from stk' l ps
    have [simp]: "take (length ps) stk = aps" by simp
```

```
    from w ps
    have widen: "list_all2 (λx y. G ⊢ x ⪯ y) apTs ps"
      by (simp add: list_all2_def)

    from stk' l ps have "length ps < length stk" by simp
    moreover
    from wf x C
    have x: "G,hp ⊢ x ::⪯ Class C" by (rule conf_widen)
    hence "isRef x" by simp
    moreover
    { assume "x ≠ Null"
      with x
      obtain D fs where
        hp: "hp (the_Addr x) = Some (D,fs)" and
        D:  "G ⊢ D ≺C C"
        by - (drule non_npD, assumption, clarsimp, blast)
      hence "hp (the_Addr x) ≠ None" (is ?P1) by simp
      moreover
      from wf mth hp D
      have "method (G, cname_of hp x) (mn, ps) ≠ None" (is ?P2)
        by (auto dest: subcls_widen_methd)
      moreover
      from aps have "list_all2 (conf G hp) aps (rev apTs)"
        by (simp add: list_all2_approx approx_stk_def approx_loc_def)
      hence "list_all2 (conf G hp) (rev aps) (rev (rev apTs))"
        by (simp only: list_all2_rev)
      hence "list_all2 (conf G hp) (rev aps) apTs" by simp
      with wf widen
      have "list_all2 (conf G hp) (rev aps) ps" (is ?P3)
        by - (rule list_all2_conf_widen)
      ultimately
      have "?P1 ∧ ?P2 ∧ ?P3" by blast
    }
    moreover
    note Invoke
    ultimately
    show ?thesis by simp
  next
    case Return with stk app phi meth frames
    show ?thesis
      apply clarsimp
      apply (drule conf_widen [OF wf], assumption)
      apply (clarsimp simp add: neq_Nil_conv isRef_def2)
      done
  qed auto
  hence "check G s" by (simp add: check_def meth)
} ultimately
have "check G s" by blast
```

```
  thus "exec_d G (Normal s) ≠ TypeError" ..
qed
```

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)"
  apply (unfold exec_all_def)
  apply (erule rtrancl_induct)
   apply (simp add: exec_all_d_def)
  apply simp
  apply (fold exec_all_def)
  apply (frule BV_correct, assumption+)
  apply (drule no_type_error, assumption, drule no_type_error_commutes, simp)
  apply (simp add: exec_all_d_def)
  apply (rule rtrancl_trans, assumption)
  apply blast
  done
```

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_jvm_prog Γ Φ" and "Γ,Φ ⊢JVM s √"
  shows "Γ ⊢ (Normal s) -jvmd→ (Normal t) = Γ ⊢ s -jvm→ t"
  by rule (erule defensive_imp_aggressive,rule welltyped_aggressive_imp_defensive)


corollary welltyped_initial_commutes:
  fixes G ("Γ") and Phi ("Φ")
  assumes "wt_jvm_prog Γ Φ"
  assumes "is_class Γ C" "method (Γ,C) (m,[]) = Some (C, b)" "m ≠ init"
  defines start: "s ≡ start_state Γ C m"
  shows "Γ ⊢ (Normal s) -jvmd→ (Normal t) = Γ ⊢ s -jvm→ t"
proof -
  have "Γ,Φ ⊢JVM s √" by (unfold start, rule BV_correct_initial)
  thus ?thesis by  - (rule welltyped_commutes)
qed

end
```

4.24 Example Welltypings

theory *BVExample = JVMListExample + BVSpecTypeSafe + JVM:*

This theory shows type correctness of the example program in section 3.6 (p. 55) by explicitly providing a welltyping. It also shows that the start state of the program conforms to the welltyping; hence type safe execution is guaranteed.

4.24.1 Setup

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

axioms
```
  distinct_classes: "list_nam ≠ test_nam"
  distinct_fields:  "val_nam ≠ next_nam"
```

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 (arbitrary, [],[])"
  by (simp add: class_def system_defs E_def)

lemma class_NullPointer [simp]:
  "class E (Xcpt NullPointer) = Some (Object, [], [])"
  by (simp add: class_def system_defs E_def)

lemma class_OutOfMemory [simp]:
  "class E (Xcpt OutOfMemory) = Some (Object, [], [])"
  by (simp add: class_def system_defs E_def)

lemma class_ClassCast [simp]:
  "class E (Xcpt ClassCast) = Some (Object, [], [])"
  by (simp add: class_def system_defs E_def)

lemma class_list [simp]:
  "class E list_name = Some list_class"
  by (simp add: class_def system_defs E_def name_defs distinct_classes [symmetric])

lemma class_test [simp]:
  "class E test_name = Some test_class"
  by (simp add: class_def system_defs E_def name_defs distinct_classes [symmetric])
```

lemma *E_classes [simp]:*
"{C. is_class E C} = {list_name, test_name, Xcpt NullPointer,
Xcpt ClassCast, Xcpt OutOfMemory, Object}"
by *(auto simp add: is_class_def class_def system_defs E_def name_defs class_defs)*

The subclass releation spelled out:

lemma *subcls1:*
"subcls1 E = {(list_name,Object), (test_name,Object), (Xcpt NullPointer, Object),
(Xcpt ClassCast, Object), (Xcpt OutOfMemory, Object)}"
apply *(simp add: subcls1_def2)*
apply *(simp add: name_defs class_defs system_defs E_def class_def)*
apply *(auto split: split_if_asm)*
done

The subclass relation is acyclic; hence its converse is well founded:

lemma *notin_rtrancl:*
"(a,b) ∈ r ⟹ a ≠ b ⟹ (⋀y. (a,y) ∉ r) ⟹ False"*
by *(auto elim: converse_rtranclE)*

lemma *acyclic_subcls1_E: "acyclic (subcls1 E)"*
apply *(rule acyclicI)*
apply *(simp add: subcls1)*
apply *(auto dest!: tranclD)*
apply *(auto elim!: notin_rtrancl simp add: name_defs distinct_classes)*
done

lemma *wf_subcls1_E: "wf ((subcls1 E)⁻¹)"*
apply *(rule finite_acyclic_wf_converse)*
apply *(simp add: subcls1)*
apply *(rule acyclic_subcls1_E)*
done

Method and field lookup:

lemma *method_Object [simp]:*
"method (E, Object) = empty"
by *(simp add: method_rec_lemma [OF class_Object wf_subcls1_E])*

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)])"
apply *(insert class_list)*
apply *(unfold list_class_def)*
apply *(drule method_rec_lemma [OF _ wf_subcls1_E])*
apply *simp*
done

lemma *method_makelist [simp]:*
"method (E, test_name) (makelist_name, []) =

```
  Some (test_name, PrimT Void, 3, 2, make_list_ins, [])"
  apply (insert class_test)
  apply (unfold test_class_def)
  apply (drule method_rec_lemma [OF _ wf_subcls1_E])
  apply simp
  done

lemma field_val [simp]:
  "field (E, list_name) val_name = Some (list_name, PrimT Integer)"
  apply (unfold field_def)
  apply (insert class_list)
  apply (unfold list_class_def)
  apply (drule fields_rec_lemma [OF _ wf_subcls1_E])
  apply simp
  done

lemma field_next [simp]:
  "field (E, list_name) next_name = Some (list_name, Class list_name)"
  apply (unfold field_def)
  apply (insert class_list)
  apply (unfold list_class_def)
  apply (drule fields_rec_lemma [OF _ wf_subcls1_E])
  apply (simp add: name_defs distinct_fields [symmetric])
  done

lemma [simp]: "fields (E, Object) = []"
   by (simp add: fields_rec_lemma [OF class_Object wf_subcls1_E])

lemma [simp]: "fields (E, Xcpt NullPointer) = []"
  by (simp add: fields_rec_lemma [OF class_NullPointer wf_subcls1_E])

lemma [simp]: "fields (E, Xcpt ClassCast) = []"
  by (simp add: fields_rec_lemma [OF class_ClassCast wf_subcls1_E])

lemma [simp]: "fields (E, Xcpt OutOfMemory) = []"
  by (simp add: fields_rec_lemma [OF class_OutOfMemory wf_subcls1_E])

lemma [simp]: "fields (E, test_name) = []"
  apply (insert class_test)
  apply (unfold test_class_def)
  apply (drule fields_rec_lemma [OF _ wf_subcls1_E])
  apply simp
  done

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::'a)
```

```
P (Suc 0)
```

...

```
P n
```

```
constdefs
  intervall :: "nat ⇒ nat ⇒ nat ⇒ bool" ("_ ∈ [_, _')")
  "x ∈ [a, b) ≡ a ≤ x ∧ x < b"

lemma pc_0: "x < n ⟹ (x ∈ [0, n) ⟹ P x) ⟹ P x"
  by (simp add: intervall_def)

lemma pc_next: "x ∈ [n0, n) ⟹ P n0 ⟹ (x ∈ [Suc n0, n) ⟹ P x) ⟹ P x"
  apply (cases "x=n0")
  apply (auto simp add: intervall_def)
  done

lemma pc_end: "x ∈ [n,n) ⟹ P x"
  by (unfold intervall_def) arith
```

4.24.2 Program structure

The program is structurally wellformed:

```
lemma wf_struct:
  "wf_prog (λG C mb. True) E" (is "wf_prog ?mb E")
proof -
  have "unique E"
    by (simp add: system_defs E_def class_defs name_defs distinct_classes)
  moreover
  have "set SystemClasses ⊆ set E" by (simp add: system_defs E_def)
  hence "wf_syscls E" by (rule wf_syscls)
  moreover
  have "wf_cdecl ?mb E ObjectC" by (simp add: wf_cdecl_def ObjectC_def)
  moreover
  have "wf_cdecl ?mb E NullPointerC"
    by (auto elim: notin_rtrancl
             simp add: wf_cdecl_def name_defs NullPointerC_def subcls1)
  moreover
  have "wf_cdecl ?mb E ClassCastC"
    by (auto elim: notin_rtrancl
             simp add: wf_cdecl_def name_defs ClassCastC_def subcls1)
  moreover
  have "wf_cdecl ?mb E OutOfMemoryC"
    by (auto elim: notin_rtrancl
             simp add: wf_cdecl_def name_defs OutOfMemoryC_def subcls1)
  moreover
```

```
  have "wf_cdecl ?mb E (list_name, list_class)"
    apply (auto elim!: notin_rtrancl
            simp add: wf_cdecl_def wf_fdecl_def list_class_def
                      wf_mdecl_def wf_mhead_def subcls1)
    apply (auto simp add: name_defs distinct_classes distinct_fields)
    done
  moreover
  have "wf_cdecl ?mb E (test_name, test_class)"
    apply (auto elim!: notin_rtrancl
            simp add: wf_cdecl_def wf_fdecl_def test_class_def
                      wf_mdecl_def wf_mhead_def subcls1)
    apply (auto simp add: name_defs distinct_classes distinct_fields)
    done
  ultimately
  show ?thesis by (simp add: wf_prog_def E_def SystemClasses_def)
qed
```

4.24.3 Welltypings

We show welltypings of the methods *append_name* in class *list_name*, and *makelist_name* in class *test_name*:

```
lemmas eff_simps [simp] = eff_def norm_eff_def xcpt_eff_def
declare appInvoke [simp del]


constdefs
  phi_append :: method_type ("φa")
  "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)]"
  apply (simp add: check_bounded_def)
  apply (simp add: nat_number append_ins_def)
  apply (rule allI, rule impI)
  apply (elim pc_end pc_next pc_0)
  apply auto
```

done

lemma *types_append [simp]: "check_types E 3 (Suc (Suc 0)) (map OK φ_a)"*
apply *(auto simp add: check_types_def phi_append_def JVM_states_unfold)*
apply *(unfold list_def)*
apply *auto*
done

lemma *wt_append [simp]:*
"wt_method E list_name [Class list_name] (PrimT Void) 3 0 append_ins
[(Suc 0, 2, 8, Xcpt NullPointer)] φ_a"
apply *(simp add: wt_method_def wt_start_def wt_instr_def)*
apply *(simp add: phi_append_def append_ins_def)*
apply *clarify*
apply *(elim pc_end pc_next pc_0)*
apply *simp*
apply *(fastsimp simp add: match_exception_entry_def sup_state_conv subcls1)*
apply *simp*
apply *simp*
apply *(fastsimp simp add: sup_state_conv subcls1)*
apply *simp*
apply *(simp add: app_def xcpt_app_def)*
apply *simp*
apply *simp*
apply *simp*
apply *(simp add: match_exception_entry_def)*
apply *(simp add: match_exception_entry_def)*
apply *simp*
apply *simp*
done

Some abbreviations for readability

syntax
Clist :: ty
Ctest :: ty
translations
"Clist" == "Class list_name"
"Ctest" == "Class test_name"

constdefs
phi_makelist :: method_type ("φ_m")
"$\varphi_m \equiv$ 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 []"
  apply (simp add: check_bounded_def)
  apply (simp add: nat_number make_list_ins_def)
  apply (rule allI, rule impI)
  apply (elim pc_end pc_next pc_0)
  apply auto
  done

lemma types_makelist [simp]: "check_types E 3 (Suc (Suc (Suc 0))) (map OK φm)"
  apply (auto simp add: check_types_def phi_makelist_def JVM_states_unfold)
  apply (unfold list_def)
  apply auto
  done

lemma wt_makelist [simp]:
  "wt_method E test_name [] (PrimT Void) 3 2 make_list_ins [] φm"
  apply (simp add: wt_method_def)
  apply (simp add: make_list_ins_def phi_makelist_def)
  apply (simp add: wt_start_def nat_number)
  apply (simp add: wt_instr_def)
  apply clarify
  apply (elim pc_end pc_next pc_0)
  apply (simp add: match_exception_entry_def)
  apply simp
  apply simp
  apply simp
  apply (simp add: match_exception_entry_def)
  apply (simp add: match_exception_entry_def)
  apply simp
  apply simp
```

```
  apply simp
  apply (simp add: match_exception_entry_def)
  apply (simp add: match_exception_entry_def)
  apply simp
  apply simp
  apply simp
  apply (simp add: match_exception_entry_def)
  apply (simp add: match_exception_entry_def)
  apply simp
  apply (simp add: app_def xcpt_app_def)
  apply simp
  apply simp
  apply simp
  apply (simp add: app_def xcpt_app_def)
  apply simp
  done
```

The whole program is welltyped:

```
constdefs
  Phi :: prog_type ("Φ")
  "Φ C sg ≡ if C = test_name ∧ sg = (makelist_name, []) then φm else
            if C = list_name ∧ sg = (append_name, [Class list_name]) then φa else []"

lemma wf_prog:
  "wt_jvm_prog E Φ"
  apply (unfold wt_jvm_prog_def)
  apply (rule wf_mb'E [OF wf_struct])
  apply (simp add: E_def)
  apply clarify
  apply (fold E_def)
  apply (simp add: system_defs class_defs Phi_def)
  apply auto
  done
```

4.24.4 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 √"
  apply (rule BV_correct_initial)
    apply (rule wf_prog)
   apply simp
  apply simp
  done
```

4.24.5 Example for code generation: inferring method types

```
constdefs
  test_kil :: "jvm_prog ⇒ cname ⇒ ty list ⇒ ty ⇒ nat ⇒ nat ⇒
               exception_table ⇒ instr list ⇒ JVMType.state list"
  "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 = (UN p:{..size ss(}. if ¬stable r step ss p then {p} else {})"
  apply (unfold unstables_def)
  apply (rule equalityI)
  apply (rule subsetI)
  apply (erule CollectE)
  apply (erule conjE)
  apply (rule UN_I)
  apply simp
  apply simp
  apply (rule subsetI)
  apply (erule UN_E)
  apply (case_tac "¬ stable r step ss p")
  apply simp+
  done
```

lemmas *[code] = lessThan_0 lessThan_Suc*

constdefs
```
  some_elem :: "'a set ⇒ 'a"
  "some_elem == (%S. SOME x. x : S)"
```

lemma *[code]:*
```
"iter f step ss w =
 while (%(ss,w). w ≠ {})
       (%(ss,w). let p = some_elem w
                 in propa f (step p (ss!p)) ss (w-{p}))
       (ss,w)"
  by (unfold iter_def some_elem_def, rule refl)
```

types_code
```
  set ("_ list")
```

consts_code
```
  "{}"     ("[]")
  "insert" ("(_ ins _)")
  "op :"   ("(_ mem _)")
  "op Un"  ("(_ union _)")
  "image"  ("map")
  "UNION"  ("(fn A => fn f => flat (map f A))")
  "Bex"    ("(fn A => fn f => exists f A)")
  "Ball"   ("(fn A => fn f => forall f A)")
  "some_elem" ("hd")
  "op -" :: "'a set ⇒ 'a set ⇒ 'a set"  ("(_ \\ _)")
```

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))))"
  apply (unfold JVMType.sup_def JVMType.sl_def Opt.esl_def Err.sl_def
         stk_esl_def reg_sl_def Product.esl_def
```

```
        Listn.sl_def upto_esl_def JType.esl_def Err.esl_def)
  by simp

lemmas [code] =
  meta_eq_to_obj_eq [OF JType.sup_def [unfolded exec_lub_def]]
  meta_eq_to_obj_eq [OF JVM_le_unfold]

lemmas [code ind] = rtrancl_refl converse_rtrancl_into_rtrancl

generate_code
  test1 = "test_kil E list_name [Class list_name] (PrimT Void) 3 0
    [(Suc 0, 2, 8, Xcpt NullPointer)] append_ins"
  test2 = "test_kil E test_name [] (PrimT Void) 3 2 [] make_list_ins"

ML test1
ML test2

end
```

Bibliography

[1] G. Klein. *Verified Java Bytecode Verification*. PhD thesis, Institut für Informatik, Technische Universität München, 2002. submitted.

[2] 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.

[3] 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.

[4] T. Nipkow. Verified bytecode verifiers. In F. Honsell, editor, *Foundations of Software Science and Computation Structures (FOSSACS 2001)*, volume 2030, pages 347–363, 2001.

[5] 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.

[6] 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.