μJava
Java Virtual Machine and Bytecode Verifier
– Object Initialization –

Gerwin Klein

November 28, 2002

Contents

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.

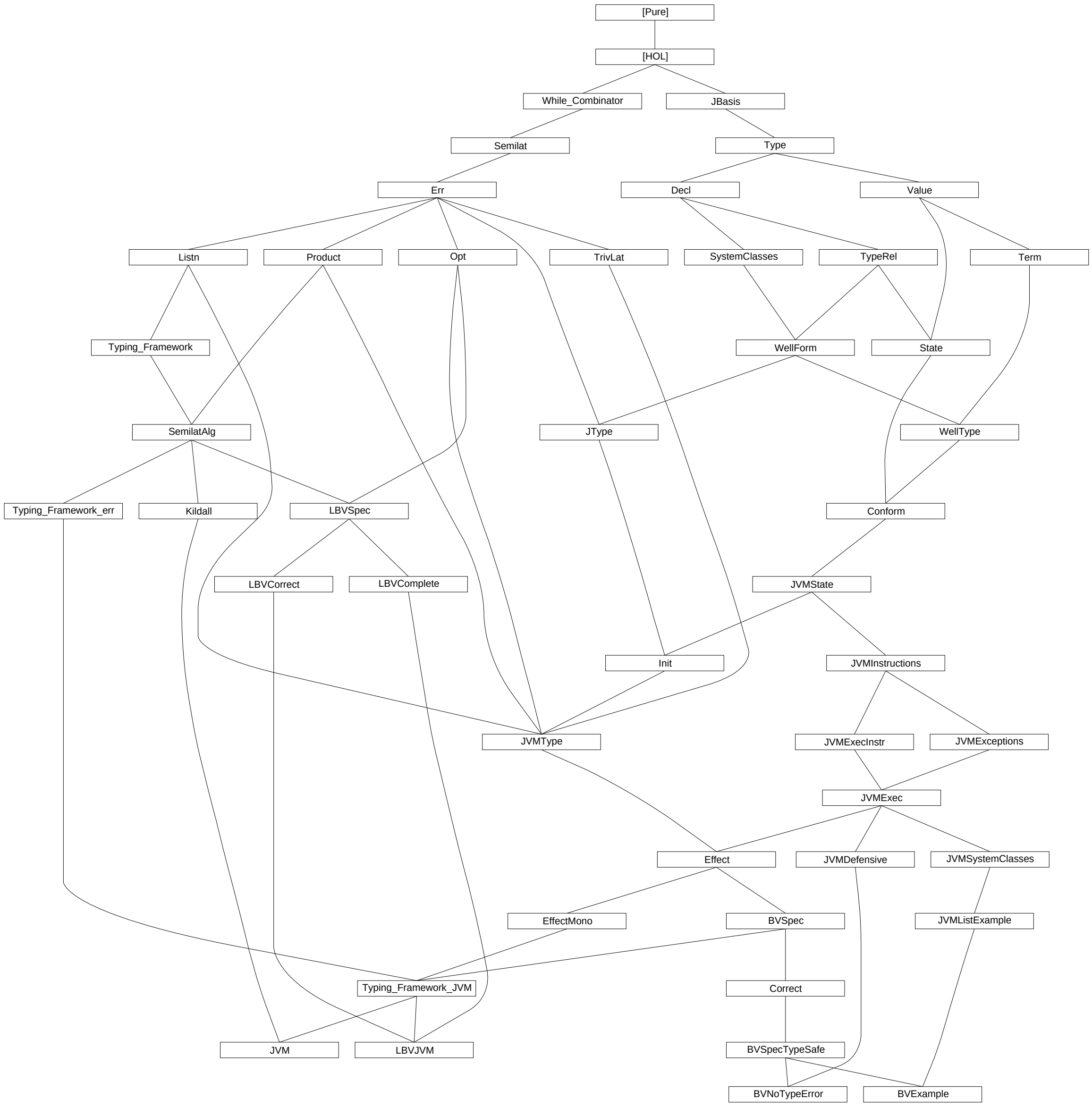

Figure 1.1: Theory Dependency Graph

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 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

consts class_rec ::"'c prog × cname ⇒
        'a ⇒ (cname ⇒ fdecl list ⇒ 'c mdecl list ⇒ 'a ⇒ 'a) ⇒ 'a"

recdef class_rec "same_fst (λG. wf ((subcls1 G)^-1)) (λG. (subcls1 G)^-1)"
      "class_rec (G,C) = (λt f. case class G C of None ⇒ arbitrary
                         | Some (D,fs,ms) ⇒ if wf ((subcls1 G)^-1) then
      f C fs ms (if C = Object then t else class_rec (G,D) t f) else arbitrary)"
(hints intro: subcls1I)

declare class_rec.simps [simp del]


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)"
  apply (rule class_rec.simps [THEN trans [THEN fun_cong [THEN fun_cong]]])
  apply simp
  done


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 [simp]:
  "G ⊢ T ≼ PrimT T' = (T = PrimT T')"
  by (rule, erule widen.elims) auto

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
```

2.5 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

2.6 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 = "xcpt option × state" — state including exception information
```

syntax
```
  heap    :: "state => aheap"
  locals  :: "state => locals"
  Norm    :: "state => xstate"
```

translations
```
  "heap"   => "fst"
  "locals" => "snd"
  "Norm s" == "(None,s)"
```

constdefs
```
  new_Addr  :: "aheap => loc × xcpt option"
 "new_Addr h == SOME (a,x). (h a = None ∧  x = None) |  x = Some OutOfMemory"

  raise_if  :: "bool => xcpt => xcpt option => xcpt option"
 "raise_if c x xo == if c ∧  (xo = None) then Some x else xo"

  np    :: "val => xcpt option => xcpt 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:*

```
"(a,x) = new_Addr h ==> h a = None ∧ x = None | x = Some OutOfMemory"
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 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 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 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 (if c then xc else NullPointer)"
apply (unfold raise_if_def)
apply (simp (no_asm))
done

end
```

2.7 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

2.8 System Classes

theory *SystemClasses = Decl:*

This theory provides definitions for the *Object* class, and the system exceptions. It leaves methods empty (to be instantiated later).

constdefs
```
  ObjectC_decl :: "'c mdecl list ⇒ 'c cdecl"
  "ObjectC_decl ms ≡ (Object, (arbitrary,[],ms))"

  NullPointerC_decl :: "'c mdecl list ⇒ 'c cdecl"
  "NullPointerC_decl ms ≡ (Xcpt NullPointer, (Object,[],ms))"

  ClassCastC_decl :: "'c mdecl list ⇒ 'c cdecl"
  "ClassCastC_decl ms ≡ (Xcpt ClassCast, (Object,[],ms))"

  OutOfMemoryC_decl :: "'c mdecl list ⇒ 'c cdecl"
  "OutOfMemoryC_decl ms ≡ (Xcpt OutOfMemory, (Object,[],ms))"

  SystemClasses :: "cname list"
  "SystemClasses ≡ [Object, Xcpt NullPointer, Xcpt ClassCast, Xcpt OutOfMemory]"
```

lemmas *SystemClass_decl_defs = ObjectC_decl_def NullPointerC_decl_def ClassCastC_decl_def OutOfMemoryC_decl_def*

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 == set SystemClasses ⊆ fst ` (set G)"

 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 SystemClasses_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 SystemClasses_def)
  apply clarify
  apply (cases x)
  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 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 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 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 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 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

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


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

inductive "ty_expr E" "ty_exprs E" "wt_stmt E" 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 wt_is_type: "wf_prog wf_mb G ⟹ ((G,L)⊢e::T ⟶ is_type G T) ∧
       ((G,L)⊢es[::]Ts ⟶ Ball (set Ts) (is_type G)) ∧ ((G,L)⊢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, COMP swap_prems_rl]

end
```

2.11 Conformity Relations for Type Soundness Proof

theory *Conform = State + WellType:*

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]"

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

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
```

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 conforms

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

```
done

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

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

lemma conforms_hext: "[|(h,l)::≼(G,lT); h≤|h'; G⊢h h'√ |] ==> (h',l)::≼(G,lT)"
apply( fast dest: conforms_localD elim!: conformsI lconf_hext)
done

lemma conforms_upd_obj:
  "[|(h,l)::≼(G, lT); G,h(a↦obj)⊢obj√; h≤|h(a↦obj)|] ==> (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:
"[|(h, l)::≼(G, lT); G,h⊢v::≼T; lT va = Some T|] ==> (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:*

For object initialization, we tag each location with the current init status. The tags use an extended type system for object initialization (that gets reused in the BV).
We have either

- usual initialized types, or
- a class that is not yet initialized and has been created by a *New* instruction at a certain line number, or
- a partly initialized class (on which the next super class constructor has to be called). We store the name of the class the super class constructor has to be called of.

datatype *init_ty = Init ty | UnInit cname nat | PartInit cname*

3.1.1 Frame Stack

types
```
opstack   = "val list"
locvars   = "val list"
p_count   = nat
ref_upd    = "(val × val)"

frame = "opstack ×
         locvars ×
         cname ×
         sig ×
         p_count ×
         ref_upd"
```

— 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
— ref update for obj init proof

3.1.2 Exceptions

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

— redefines State.new_Addr:
```
new_Addr :: "aheap ⇒ loc × val option"
"new_Addr h ≡ let (a, x) = State.new_Addr h
              in  (a, raise_system_xcpt (x ~= None) OutOfMemory)"
```

types
```
init_heap = "loc ⇒ init_ty"
```

— type tags to track init status of objects

```
jvm_state = "val option × aheap × init_heap × frame list"
```
— exception flag, heap, tag heap, frames

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))"

  start_iheap :: init_heap
  "start_iheap ≡ (((λx. arbitrary)
                 (XcptRef NullPointer := Init (Class (Xcpt NullPointer))))
                 (XcptRef ClassCast := Init (Class (Xcpt ClassCast))))
                 (XcptRef OutOfMemory := Init (Class ((Xcpt OutOfMemory))))"
```

3.1.3 Lemmas

```
lemma new_AddrD:
  assumes new: "new_Addr hp = (ref, xcp)"
  shows "hp ref = None ∧ xcp = None ∨ xcp = Some (Addr (XcptRef OutOfMemory))"
proof -
  from new obtain xcpT where old: "State.new_Addr hp = (ref,xcpT)"
    by (cases "State.new_Addr hp") (simp add: new_Addr_def)
  from this [symmetric]
  have "hp ref = None ∧ xcpT = None ∨ xcpT = Some OutOfMemory"
    by (rule State.new_AddrD)
  with new old show ?thesis
    by (auto simp add: new_Addr_def raise_system_xcpt_def)
qed


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
        | Invoke_special cname mname "ty list"
                                        — no dynamic type lookup, for constructors
        | Return                        — return from method
        | Pop                           — pop top element from opstack
        | Dup                           — duplicate top element of opstack
        | Dup_x1                        — duplicate next to top element
        | Dup_x2                        — duplicate 3rd element
        | 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"
  jvm_prog = "jvm_method prog"
```

end

3.3 JVM Instruction Semantics

theory *JVMExecInstr = JVMInstructions + JVMState:*

the method name of constructors:

```
consts
  init :: mname
```

replace a by b in l:

```
constdefs
  replace :: "'a ⇒ 'a ⇒ 'a list ⇒ 'a list"
  "replace a b l == map (λx. if x = a then b else x) l"
```

some lemmas about replace

```
lemma replace_removes_elem:
  "a ≠ b ⟹ a ∉ set (replace a b l)"
  by (unfold replace_def) auto

lemma replace_length [simp]:
  "length (replace a b l) = length l" by (simp add: replace_def)

lemma replace_Nil [iff]:
  "replace x y [] = []" by (simp add: replace_def)

lemma replace_Cons:
  "replace x y (l#ls) = (if l = x then y else l)#(replace x y ls)"
  by (simp add: replace_def)

lemma replace_map:
  "inj f ==> replace (f x) (f y) (map f l) = map f (replace x y l)"
apply (induct l)
 apply (simp add: replace_def)
apply (simp add: replace_def)
apply clarify
apply (drule injD, assumption)
apply simp
done

lemma replace_id:
  "x ∉ set l ∨ x = y ⟹ replace x y l = l"
  apply (induct l)
  apply (auto simp add: replace_def)
  done
```

single execution step for each instruction:

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

```
"exec_instr (Load idx) G hp ihp stk vars Cl sig pc z frs =
     (None, hp, ihp, ((vars ! idx) # stk, vars, Cl, sig, pc+1, z)#frs)"

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

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

"exec_instr (New C) G hp ihp stk vars Cl sig pc z frs =
       (let (oref,xp') = new_Addr hp;
            hp'        = if xp'=None then hp(oref ↦ blank G C) else hp;
      ihp' = if xp'=None then ihp(oref := UnInit C pc) else ihp;
            stk'       = if xp'=None then (Addr oref)#stk else stk;
      pc'  = if xp'=None then pc+1 else pc
        in
     (xp', hp', ihp', (stk', vars, Cl, sig, pc', z)#frs))"

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

"exec_instr (Putfield F C) G hp ihp stk vars Cl sig pc z 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', ihp, (tl (tl stk), vars, Cl, sig, pc', z)#frs))"

"exec_instr (Checkcast C) G hp ihp stk vars Cl sig pc z 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, ihp, (stk', vars, Cl, sig, pc', z)#frs))"


"exec_instr (Invoke C mn ps) G hp ihp stk vars Cl sig pc z frs =
       (let n          = length ps;
      args = take n stk;
            oref = stk!n;
            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
```

```
           [([],oref#(rev args)@(replicate mxl arbitrary),dc,(mn,ps),0,arbitrary)]
                 else [])
        in
      (xp', hp, ihp, frs'@(stk, vars, Cl, sig, pc, z)#frs))"
```

— Because exception handling needs the pc of the Invoke instruction,
— Invoke doesn't change *stk* and *pc* yet (*Return* does that).

```
"exec_instr (Invoke_special C mn ps) G hp ihp stk vars Cl sig pc z frs =
       (let n          = length ps;
      args = take n stk;
            oref       = stk!n;
      addr = the_Addr oref;
            x'   = raise_system_xcpt (oref=Null) NullPointer;
            dynT       = fst(the(hp addr));
            (dc,mh,mxs,mxl,c)= the (method (G,C) (mn,ps));
      (addr',x'') = new_Addr hp;
      xp'  = if x' = None then x'' else x';
      hp'  = if xp' = None then hp(addr' ↦ blank G dynT) else hp;
      ihp' = if C = Object then ihp(addr':= Init (Class dynT))
             else ihp(addr' := PartInit C);
      ihp'' = if xp' = None then ihp' else ihp;
      z'   = if C = Object then (Addr addr', Addr addr') else (Addr addr', Null);
            frs'       = (if xp'=None then
             [([],(Addr addr')#(rev args)@(replicate mxl arbitrary),dc,(mn,ps),0,z')]
                 else [])
        in
      (xp', hp', ihp'', frs'@(stk, vars, Cl, sig, pc, z)#frs))"

"exec_instr Return G hp ihp stk0 vars Cl sig0 pc z0 frs =
       (if frs=[] then
    (None, hp, ihp, [])
  else let
    val      = hd stk0;
    (mn,pt) = sig0;
    (stk,loc,C,sig,pc,z) = hd frs;
    (b,c)    = z0;
    (a,c')   = z;
    n        = length pt;
    args     = take n stk;
    addr     = stk!n;
    drpstk   = drop (n+1) stk;
    stk'     = if mn=init then val#replace addr c drpstk else val#drpstk;
    loc'     = if mn=init then replace addr c loc else loc;
    z'       = if mn=init ∧ z = (addr,Null) then (a,c) else z
        in
      (None, hp, ihp, (stk',loc',C,sig,pc+1,z')#tl frs))"
```

— Return drops arguments from the caller's stack and increases
— the program counter in the caller

— *z* is only updated if we are in a constructor and have initialized the
— same reference as the constructor in the frame above (otherwise we are
— in the last constructor of the init chain)

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

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

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

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

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

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

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

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

"exec_instr Throw G hp ihp stk vars Cl sig pc z frs =
 (let xcpt  = raise_system_xcpt (hd stk = Null) NullPointer;
      xcpt' = if xcpt = None then Some (hd stk) else xcpt
  in
     (xcpt', hp, ihp, (stk, vars, Cl, sig, pc, z)#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
  cname_of :: "aheap ⇒ val ⇒ cname"
  ex_table_of :: "jvm_method ⇒ exception_table"

translations
  "cname_of hp v" == "fst (the (hp (the_Addr v)))"
  "ex_table_of m" == "snd (snd (snd m))"


consts
  find_handler :: "jvm_prog ⇒ val option ⇒ aheap ⇒ init_heap ⇒ frame list ⇒ jvm_state"
primrec
  "find_handler G xcpt hp ihp [] = (xcpt, hp, ihp, [])"
  "find_handler G xcpt hp ihp (fr#frs) =
      (case xcpt of
         None ⇒ (None, hp, ihp, fr#frs)
       | Some xc ⇒
       let (stk,loc,C,sig,pc,r) = 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 ihp frs
        | Some handler_pc ⇒ (None, hp, ihp, ([xc], loc, C, sig, handler_pc, r)#frs)))"
```

Expresses that a value is tagged with an initialized type (only applies to addresses and then only if the heap contains a value for the address)

```
constdefs
  is_init :: "aheap ⇒ init_heap ⇒ val ⇒ bool"
```

```
  "is_init hp ih v ≡
  ∀loc. v = Addr loc ⟶ hp loc ≠ None ⟶ (∃t. ih loc = Init t)"
```

System exceptions are allocated in all heaps.

```
constdefs
  preallocated :: "aheap ⇒ init_heap ⇒ bool"
  "preallocated hp ihp ≡ ∀x. ∃fs. hp (XcptRef x) = Some (Xcpt x, fs) ∧ is_init hp ihp
(Addr (XcptRef x))"

lemma preallocatedD [simp,dest]:
  "preallocated hp ihp ⟹ ∃fs. hp (XcptRef x) = Some (Xcpt x, fs) ∧ is_init hp ihp (Addr
(XcptRef x))"
  by (unfold preallocated_def) fast

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

lemma cname_of_xcp:
  "raise_system_xcpt b x = Some xcp ⟹ preallocated hp ihp
  ⟹ cname_of hp xcp = Xcpt x"
proof -
  assume "raise_system_xcpt b x = Some xcp"
  hence "xcp = Addr (XcptRef x)"
    by (simp add: raise_system_xcpt_def split: split_if_asm)
  moreover
  assume "preallocated hp ihp"
  then obtain fs where "hp (XcptRef x) = Some (Xcpt x, fs)" ..
  ultimately show ?thesis by simp
qed


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

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

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

— recdef only used for pattern matching

recdef *exec "{}"*
```
  "exec (G, xp, hp, ihp, []) = None"

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

  "exec (G, Some xp, hp, ihp, 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,ins,et) = the (method (G,C) (m,[])) in
  (None, start_heap G, start_iheap,
   [([], Null # replicate mxl arbitrary, C, (m,[]), 0, (Null,Null))])"
```

end

3.6 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"
primrec
  "isAddr Unit       = False"
  "isAddr Null       = False"
  "isAddr (Bool b)   = False"
  "isAddr (Intg i)   = False"
  "isAddr (Addr loc) = True"

consts isIntg :: "val ⇒ bool"
primrec
  "isIntg Unit       = False"
  "isIntg Null       = False"
  "isIntg (Bool b)   = False"
  "isIntg (Intg i)   = True"
  "isIntg (Addr loc) = False"

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



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

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

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

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

  "check_instr (Getfield F C) G hp ihp stk vars Cl sig pc z maxpc 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 ∧ is_init hp ihp ref ∧
    (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 ihp stk vars Cl sig pc z maxpc 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 ∧ is_init hp ihp v ∧ isRef ref ∧ (ref ≠ Null ⟶
    hp (the_Addr ref) ≠ None ∧ is_init hp ihp ref ∧
    (let (D,vs) = the (hp (the_Addr ref)) in
      G ⊢ D ≼C C ∧ G,hp ⊢ v ::≼ T))))"

"check_instr (Checkcast C) G hp ihp stk vars Cl sig pc z maxpc frs =
(0 < length stk ∧ is_class G C ∧ isRef (hd stk) ∧ is_init hp ihp (hd stk))"


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


"check_instr (Invoke_special C mn ps) G hp ihp stk vars Cl sig pc z maxpc frs =
(length ps < length stk ∧ mn = init ∧
(let n = length ps; ref = stk!n in
isRef ref ∧ (ref ≠ Null ⟶
  hp (the_Addr ref) ≠ None ∧
  method (G,C) (mn,ps) ≠ None ∧
  fst (the (method (G,C) (mn,ps))) = C ∧
  list_all2 (λv T. G,hp ⊢ v ::≼ T ∧ is_init hp ihp v) (rev (take n stk)) ps) ∧
  (case ihp (the_Addr ref) of
     Init T ⇒ False
   | UnInit C' pc' ⇒ C' = C
   | PartInit C' ⇒ C' = Cl ∧ G ⊢ C' ≺C1 C)
))"

"check_instr Return G hp ihp stk0 vars Cl sig0 pc z0 maxpc 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 ∧ is_init hp ihp v) ∧
  (fst sig0 = init ⟶
    snd z0 ≠ Null ∧ isRef (snd z0) ∧ is_init hp ihp (snd z0))))"

"check_instr Pop G hp ihp stk vars Cl sig pc z maxpc frs =
(0 < length stk)"

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

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

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

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

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

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

"check_instr (Goto b) G hp ihp stk vars Cl sig pc z maxpc frs =
(0 ≤ int pc+b ∧ nat(int pc+b) < maxpc)"

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

constdefs

```
check :: "jvm_prog ⇒ jvm_state ⇒ bool"
"check G s ≡ let (xcpt, hp, ihp, frs) = s in
             (case frs of [] ⇒ True | (stk,loc,C,sig,pc,z)#frs' ⇒
              (let ins = fifth (the (method (G,C) sig)); i = ins!pc in
               check_instr i G hp ihp stk loc C sig pc z (length ins) 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
```

3.7 System Class Implementations (JVM)

theory *JVMSystemClasses = WellForm + JVMExec:*

This theory provides bytecode implementations for the system class library. Each class has a default constructor.

```
constdefs
  Object_ctor :: "jvm_method mdecl"
  "Object_ctor ≡ ((init, []), PrimT Void, (1, 0, [LitPush Unit, Return], []))"

  ObjectC :: "jvm_method cdecl"
  "ObjectC ≡ ObjectC_decl [Object_ctor]"

  Default_ctor :: "jvm_method mdecl"
  "Default_ctor ≡
  ((init, []), PrimT Void, (1,0,[Load 0, Invoke_special Object init [], Return], []))"

  NullPointerC :: "jvm_method cdecl"
  "NullPointerC ≡ NullPointerC_decl [Default_ctor]"

  ClassCastC :: "jvm_method cdecl"
  "ClassCastC ≡ ClassCastC_decl [Default_ctor]"

  OutOfMemoryC :: "jvm_method cdecl"
  "OutOfMemoryC ≡ OutOfMemoryC_decl [Default_ctor]"

  JVMSystemClasses :: "jvm_method cdecl list"
  "JVMSystemClasses ≡ [ObjectC, NullPointerC, ClassCastC, OutOfMemoryC]"

lemmas SystemClassC_defs = SystemClass_decl_defs ObjectC_def NullPointerC_def
  OutOfMemoryC_def ClassCastC_def Default_ctor_def Object_ctor_def

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

lemma wf_syscls:
  "set JVMSystemClasses ⊆ set G ⟹ wf_syscls G"
  apply (unfold wf_syscls_def SystemClasses_def JVMSystemClasses_def SystemClassC_defs)
  apply (drule fst_mono)
  apply simp
  done

end
```

3.8 Example for generating executable code from JVM semantics

```
theory JVMListExample = JVMSystemClasses + 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)],
     [Default_ctor,
      ((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,
        Dup,
        Invoke_special list_name init [],
        Pop,
        Store 0,
        LitPush (Intg 1),
        Putfield val_name list_name,
        New list_name,
        Dup,
        Invoke_special list_name init [],
        Pop,
        Dup,
        Store 1,
        LitPush (Intg 2),
        Putfield val_name list_name,
        New list_name,
        Dup,
        Invoke_special list_name init [],
        Pop,
        Dup,
        Store 2,
        LitPush (Intg 3),
        Putfield val_name list_name,
        Load 0,
        Load 1,
        Invoke list_name append_name [Class list_name],
        Load 0,
        Load 2,
        Invoke list_name append_name [Class list_name],
        Return]"

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

  E :: jvm_prog
  "E == JVMSystemClasses @ [(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 *}")

  "wf" ("true?")

  "arbitrary" ("(raise ERROR)")
  "arbitrary" :: "val × val" ("{* (Unit,Unit) *}")
```

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

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

```
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;
*}
```

3.8.1 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) *}
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

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* $\wedge$ *closed {Err} f* $\wedge$ *Err* $\sqcup_f$ *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 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
```

4.8 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
```

4.9 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
```

4.10 Java Type System with Object Initialization

theory *Init = JType + JVMState:*

init_ty is still a (error) semilattice:

```
constdefs
  init_tys :: "'c prog ⇒ init_ty set"
  "init_tys G == {x. ∃y ∈ (fst (JType.esl G)). x = Init y} ∪
                {x. ∃c n. x = UnInit c n} ∪ {x. ∃c. x = PartInit c}"

  init_le :: "'c prog ⇒ init_ty ord" ("_ ⊢ _ ≼i _" [71,71] 70)
  "G ⊢ a ≼i b == case a of
                   Init t1 ⇒ (case b of Init t2 ⇒ fst (snd (JType.esl G)) t1 t2
                                      | UnInit c n ⇒ False
                                      | PartInit c ⇒ False)
                 | UnInit c1 n1 ⇒ a = b
                 | PartInit c1 ⇒ a = b"

  sup :: "'c prog ⇒ init_ty ⇒ init_ty ⇒ init_ty err"
  "sup G a b ==
  case a of
    Init t1 ⇒ (case b of Init t2 ⇒ (case snd (snd (JType.esl G)) t1 t2 of
                                    Err ⇒ Err | OK x ⇒ OK (Init x))
                       | UnInit c n ⇒ Err
                       | PartInit c ⇒ Err)
  | UnInit c1 n1 ⇒ (case b of Init t ⇒ Err
                            | UnInit c2 n2 ⇒ (if a=b then OK a else Err)
                            | PartInit c2 ⇒ Err)
  | PartInit c1 ⇒ (case b of Init t ⇒ Err
                            | UnInit c2 n2 ⇒ Err
                            | PartInit c2 ⇒ (if a=b then OK a else Err))"

  esl :: "'c prog ⇒ init_ty esl"
  "esl G == (init_tys G, init_le G, sup G)"

syntax (HTML)
  init_le :: "['code prog,init_ty,init_ty] ⇒ bool" ("_ |- _ <=i _")

lemma le_Init_Init [iff]:
  "init_le G (Init t1) (Init t2) = subtype G t1 t2"
  by (unfold init_le_def JType.esl_def) simp

lemma le_Init_UnInit [iff]:
  "init_le G (Init t) (UnInit c n) = False"
  by (unfold init_le_def) simp

lemma le_UnInit_Init [iff]:
  "init_le G (UnInit c n) (Init t) = False"
  by (unfold init_le_def) simp

lemma le_UnInit_UnInit [iff]:
  "init_le G (UnInit c1 n1) (UnInit c2 n2) = (c1 = c2 ∧ n1 = n2)"
  by (unfold init_le_def) simp
```

```
lemma init_le_Init:
  "(G ⊢ Init t ≼i x) = (∃t'. x = Init t' ∧ G ⊢ t ≼ t')"
  by (simp add: init_le_def JType.esl_def subtype_def split: init_ty.splits)

lemma init_le_Init2:
  "(G ⊢ x ≼i Init t') = (∃t. x = Init t ∧ G ⊢ t ≼ t')"
  by (cases x, auto simp add: init_le_def JType.esl_def subtype_def)

lemma init_le_UnInit [iff]:
  "(G ⊢ UnInit C pc ≼i x) = (x = UnInit C pc)"
  by (cases x, auto simp add: init_le_def)

lemma init_le_UnInit2 [iff]:
  "G ⊢ T ≼i UnInit C pc = (T = UnInit C pc)"
  by (cases T, auto simp add: init_le_def)

lemma init_le_PartInit [iff]:
  "(G ⊢ PartInit C ≼i x) = (x = PartInit C)"
  by (cases x, auto simp add: init_le_def)

lemma init_le_PartInit2 [iff]:
  "(G ⊢ x ≼i PartInit C) = (x = PartInit C)"
  by (cases x, auto simp add: init_le_def)

lemma init_refl [simp]: "init_le G x x"
  by (auto simp add: init_le_def JType.esl_def subtype_def split: init_ty.split)

lemma init_trans [trans]:
  "⟦ init_le G a b; init_le G b c ⟧ ⟹ init_le G a c"
  by (auto intro: widen_trans
           simp add: init_le_def JType.esl_def subtype_def
           split: init_ty.splits)

lemma ac_order_init:
  "acyclic (subcls1 G) ⟹ order (init_le G)"
proof -
  assume "acyclic (subcls1 G)"
  hence "order (subtype G)"
    by (rule order_widen)
  hence [simp]: "⋀x y. ⟦ init_le G x y; init_le G y x ⟧ ⟹ x = y"
    apply (unfold init_le_def JType.esl_def)
    apply (simp split: init_ty.splits)
    apply (unfold order_def lesub_def)
    apply blast
    done

  show ?thesis
    by (unfold order_def lesub_def) (auto intro: init_trans)
qed

lemma order_init:
  "wf_prog wf_mb G ⟹ order (init_le G)"
  by (drule acyclic_subcls1, rule ac_order_init)
```

```
lemma wf_converse_subcls1_impl_acc_init:
  "wf ((subcls1 G)⁻¹) ⟹ acc (init_le G)"
proof -
  assume "wf ((subcls1 G)⁻¹)"
  hence "acc (subtype G)"
    by (rule wf_converse_subcls1_impl_acc_subtype)
  hence s: "∀Q. (∃x. x ∈ Q) ⟶ (∃z∈Q. ∀y. subtype G z y ∧ z ≠ y ⟶ y ∉ Q)"
    by (simp add: acc_def lesub_def le_def lesssub_def wf_eq_minimal)
  { fix Q x assume not_empty: "x ∈ (Q::init_ty set)"
    let ?Q = "{t. Init t ∈ Q}"
    have "∃z∈Q. ∀y. (G ⊢ z ≼i y) ∧ z ≠ y ⟶ y ∉ Q"
    proof (cases "∃t. Init t ∈ Q")
      case False with not_empty
      show ?thesis
        by (simp add: init_le_def JType.esl_def split: init_ty.split) blast
    next
      case True
      hence "∃t. t ∈ ?Q" by simp
      with s obtain z where
        "z∈?Q"  "∀y. subtype G z y ∧ z ≠ y ⟶ y ∉ ?Q"
        by - (erule allE, erule impE, blast+)
      with not_empty
      show ?thesis
        by (simp add: init_le_def JType.esl_def split: init_ty.split) blast
    qed
  }
  thus ?thesis
    by (simp add: acc_def lesub_def le_def lesssub_def wf_eq_minimal)
qed

lemma err_closed_init:
  "wf_prog wf_mb G ⟹ closed (err (init_tys G)) (lift2 (sup G))"
proof -
  assume wf: "wf_prog wf_mb G"
  then obtain
    "single_valued (subcls1 G)"
    "acyclic (subcls1 G)"
    by (blast dest: acyclic_subcls1 single_valued_subcls1)

  with wf
  have "closed (err (types G)) (lift2 (JType.sup G))"
    by (rule closed_err_types)
  hence [intro?]:
    "⋀x y. ⟦ x∈err (types G); y∈err (types G) ⟧
    ⟹ lift2 (JType.sup G) x y ∈ err (types G)"
    by (unfold closed_def plussub_def) blast

  { fix t1 t2
    assume "Init t1 ∈ init_tys G" "Init t2 ∈ init_tys G"
    then obtain
      "OK t1 ∈ err (types G)" "OK t2 ∈ err (types G)"
      by (unfold init_tys_def JType.esl_def) simp
    hence "lift2 (JType.sup G) (OK t1) (OK t2) ∈ err (types G)" ..
    moreover
```

```
    fix t assume "JType.sup G t1 t2 = OK t"
    ultimately
    have "t ∈ types G"
      by (unfold lift2_def) simp
    hence "Init t ∈ init_tys G"
      by (unfold init_tys_def JType.esl_def) simp
  } note this [intro]

  show ?thesis
    by (unfold closed_def plussub_def lift2_def sup_def JType.esl_def)
       (auto split: err.split init_ty.split)
qed


lemma err_semilat_init:
  "wf_prog wf_mb G ⟹ err_semilat (esl G)"
proof -
  assume wf: "wf_prog wf_mb G"
  moreover
  from wf obtain sa:
    "single_valued (subcls1 G)"
    "acyclic (subcls1 G)"
    by (blast dest: acyclic_subcls1 single_valued_subcls1)
  with wf
  have [simp]: "⋀x y c. ⟦ x ∈ types G; y ∈ types G; JType.sup G x y = OK c ⟧
    ⟹ subtype G x c ∧ subtype G y c"
    by (simp add: sup_subtype_greater)
  { fix x y
    assume err: "x∈err (init_tys G)" "y∈err (init_tys G)"
    { fix a b c
      assume ok: "x = OK a" "y = OK b"
      moreover
      from ok err
      have "a ∈ init_tys G ∧ b ∈ init_tys G"
        by blast
      moreover
      assume "sup G a b = OK c"
      ultimately
      have "init_le G a c ∧ init_le G b c"
        by (unfold init_le_def sup_def JType.esl_def init_tys_def)
           (auto split: init_ty.splits split_if_asm err.splits)
    }
    hence "Err.le (init_le G) x (lift2 (sup G) x y) ∧
           Err.le (init_le G) y (lift2 (sup G) x y)"
      by (simp add: Err.le_def lift2_def lesub_def split: err.split)
  }
  moreover
  { have [intro]:
      "⋀a b c. ⟦ init_le G a c; init_le G b c; sup G a b = Err ⟧ ⟹ False"
      by (unfold sup_def init_le_def JType.esl_def)
         (auto intro: sup_exists split: init_ty.split_asm err.split_asm)
    { fix x y z s
      assume "x ∈ init_tys G" "y ∈ init_tys G" "z ∈ init_tys G"
             "init_le G x z" "init_le G y z" "sup G x y = OK s"
```

```
      with wf sa
      have "init_le G s z"
        by (unfold init_le_def sup_def init_tys_def JType.esl_def)
           (auto intro: sup_subtype_smallest split: init_ty.split_asm err.split_asm)
    } note this [intro]
    fix x y z
    assume "x∈err (init_tys G)" "y∈err (init_tys G)" "z∈err (init_tys G)"
           "Err.le (init_le G) x z" "Err.le (init_le G) y z"
    hence "Err.le (init_le G) (lift2 (sup G) x y) z"
      by (unfold lesub_def plussub_def Err.le_def lift2_def JType.esl_def)
         (auto split: err.splits)
  }
  ultimately
  show ?thesis
    by (unfold semilat_def Err.sl_def esl_def plussub_def lesub_def)
       (simp add: err_closed_init order_init)
qed

end
The trivial semilattice theory TrivLat = Err:

constdefs
  esl :: "'a esl"
  "esl ≡ (UNIV, op =, λa b. if a=b then OK a else Err)"

lemma eq_order:
  "order (op =)"
  by (unfold order_def lesub_def) blast

lemma bool_err_semilat:
  "err_semilat esl"
  apply (unfold esl_def sl_def semilat_def)
  apply (simp add: eq_order)
  apply (auto simp add: closed_def plussub_def lesub_def Err.le_def lift2_def
              split: err.split)
  done

lemma [intro!]: "acc (op =)"
  by (simp add: acc_def lesssub_def lesub_def wf_def)

end
```

4.11 The JVM Type System as Semilattice

theory *JVMType = Opt + Product + Listn + Init + TrivLat:*

types
```
  locvars_type = "init_ty err list"
  opstack_type = "init_ty list"
  state_type   = "opstack_type × locvars_type"

  state_bool   = "state_type × bool"
  state        = "state_bool option err"     — for Kildall
  method_type  = "state_bool option list"    — for BVSpec

  class_type   = "sig ⇒ method_type"
  prog_type    = "cname ⇒ class_type"
```

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

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

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

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,init_ty err,init_ty err] ⇒ bool"
                ("_ |- _ <=o _" [71,71] 70)
  "sup_ty_opt G == Err.le (init_le 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 (init_le G)) (sup_loc G)"

  sup_state_bool :: "['code prog,state_bool,state_bool] ⇒ bool"
```

```
              ("_ |- _ <=b _"  [71,71] 70)
  "sup_state_bool G == Product.le (sup_state G) (op =)"

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

syntax *(xsymbols)*

```
  sup_ty_opt     :: "['code prog,init_ty err,init_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_bool :: "['code prog,state_bool,state_bool] ⇒ bool"
                  ("_ ⊢ _ <=b _" [71,71] 70)
  sup_state_opt :: "['code prog,state_type option,state_type option] ⇒ bool"
                  ("_ ⊢ _ <=' _" [71,71] 70)
```

lemma *UNIV_bool: "UNIV = {True,False}"*
  by *blast*

lemma *JVM_states_unfold:*

```
  "states G maxs maxr == err(opt(((Union {list n (init_tys G) |n. n <= maxs}) <*>
                                  list maxr (err(init_tys G))) <*> {True,False}))"
  by (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 Init.esl_def Err.esl_def TrivLat.esl_def)
     (simp add: UNIV_bool)
```

lemma *JVM_le_unfold:*

```
 "le G m n ==
  Err.le(Opt.le(Product.le(Product.le(Listn.le(init_le G))
  (Listn.le(Err.le(init_le G))))(op =)))"
  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 Init.esl_def Err.esl_def TrivLat.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 sup_state_bool_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_state_bool_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 (init_le G) a b = G ⊢ map OK a <=l map OK b"
proof
  assume "Listn.le (init_le G) a b"

  hence le: "list_all2 (init_le 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 "init_le 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 (init_le 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 init_le_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 sup_state_bool_def Product.le_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' ≼i ty)"
  by (simp add: sup_ty_opt_def Err.le_def lesub_def)

lemma sup_ty_opt_OK:
  "G ⊢ x <=o OK y = (∃x'. x = OK x' ∧ G ⊢ x' ≼i y)"
  by (cases x, auto)

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 (Init (PrimT p)) <=o X) = (X=Err ∨ X = OK (Init (PrimT p)))"
  by (auto simp add: sup_ty_opt_def Err.le_def lesub_def init_le_def
                     subtype_def JType.esl_def
              split: err.splits init_ty.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)


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::init_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 ≼i 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 ≼i 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 <=b 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 <=b 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 <=b b))"
  by (simp add: sup_state_opt_def Opt.le_def lesub_def split: option.split)
```

```
lemma sup_state_bool_conv[iff]:
  "(G ⊢ (a,b) <=b (c,d)) = ((G ⊢ a <=s c) ∧ b = d)"
  by (simp add: sup_state_bool_def Product.le_def lesub_def)


theorem sup_ty_opt_trans [trans]:
  "⟦G ⊢ a <=o b; G ⊢ b <=o c⟧ ⟹ G ⊢ a <=o c"
  by (auto intro: init_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_bool_trans [trans]:
  "⟦G ⊢ a <=b b; G ⊢ b <=b c⟧ ⟹ G ⊢ a <=b c"
  by (auto intro: sup_state_trans
           simp add: sup_state_bool_def 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 + JVMExec:*

types
```
  succ_type = "(p_count × state_bool 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 (Invoke_special C mn fpTs) pc
                               = [pc+1]"
  "succs Throw pc              = [pc]"
```

consts *theClass :: "init_ty ⇒ ty"*
primrec
```
  "theClass (PartInit C)  = Class C"
  "theClass (UnInit C pc) = Class C"
```

Effect of instruction on the state type:

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

recdef *eff' "{}"*
```
"eff' (Load idx,  G, pc, (ST, LT))          = (ok_val (LT ! idx) # ST, LT)"
"eff' (Store idx, G, pc, (ts#ST, LT))       = (ST, LT[idx:= OK ts])"
"eff' (LitPush v, G, pc, (ST, LT))          = (Init (the (typeof (λv. None) v))#ST, LT)"
"eff' (Getfield F C, G, pc, (oT#ST, LT))    = (Init (snd (the (field (G,C) F)))#ST, LT)"
"eff' (Putfield F C, G, pc, (vT#oT#ST, LT)) = (ST,LT)"
```

```
"eff' (New C, G, pc, (ST,LT))              = (UnInit C pc # ST, replace (OK (UnInit C
pc)) Err LT)"
"eff' (Checkcast C,G,pc,(Init (RefT t)#ST,LT)) = (Init (Class C) # ST,LT)"
"eff' (Pop, G, pc, (ts#ST,LT))             = (ST,LT)"
"eff' (Dup, G, pc, (ts#ST,LT))             = (ts#ts#ST,LT)"
"eff' (Dup_x1, G, pc, (ts1#ts2#ST,LT))     = (ts1#ts2#ts1#ST,LT)"
"eff' (Dup_x2, G, pc, (ts1#ts2#ts3#ST,LT)) = (ts1#ts2#ts3#ts1#ST,LT)"
"eff' (Swap, G, pc, (ts1#ts2#ST,LT))       = (ts2#ts1#ST,LT)"
"eff' (IAdd, G, pc, (t1#t2#ST,LT))         = (Init (PrimT Integer)#ST,LT)"
"eff' (Ifcmpeq b, G, pc, (ts1#ts2#ST,LT))  = (ST,LT)"
"eff' (Goto b, G, pc, s)                   = s"
```
— Return has no successor instruction in the same method:
```
"eff' (Return, G, pc, s)                   = s"
```
— Throw always terminates abruptly:
```
"eff' (Throw, G, pc, s)                    = s"
"eff' (Invoke C mn fpTs, G, pc, (ST,LT)) =
  (let ST'  = drop (length fpTs) ST;
       X    = hd ST';
       ST'' = tl ST';
       rT   = fst (snd (the (method (G,C) (mn,fpTs))))
   in ((Init rT)#ST'', LT))"
"eff' (Invoke_special C mn fpTs, G, pc, (ST,LT)) =
  (let ST'  = drop (length fpTs) ST;
       X    = hd ST';
       N    = Init (theClass X);
       ST'' = replace X N (tl ST');
       LT'  = replace (OK X) (OK N) LT;
       rT   = fst (snd (the (method (G,C) (mn,fpTs))))
   in ((Init rT)#ST'', LT'))"
```

For *Invoke_special* only: mark when invoking a constructor on a partly initialized class. app will check that we call the right constructor.

constdefs
```
  eff_bool :: "instr ⇒ jvm_prog ⇒ p_count ⇒ state_bool ⇒ state_bool"
  "eff_bool i G pc == λ((ST,LT),z). (eff'(i,G,pc,(ST,LT)),
  if ∃C p D. i = Invoke_special C init p ∧ ST!length p = PartInit D then True else z)"
```

For exception handling:

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 ⇒ cname ⇒ p_count ⇒ exception_table ⇒ cname list"
primrec
  "match G X pc [] = []"
  "match G X pc (e#es) =
  (if match_exception_entry G X pc e then [X] else match G X pc es)"

lemma match_some_entry:
  "match G X pc et = (if ∃e ∈ set et. match_exception_entry G X pc e then [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 (Xcpt NullPointer) pc et"
  "xcpt_names (Putfield F C, G, pc, et) = match G (Xcpt NullPointer) pc et"
  "xcpt_names (New C, G, pc, et)        = match G (Xcpt OutOfMemory) pc et"
  "xcpt_names (Checkcast C, G, pc, et)  = match G (Xcpt 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 (Invoke_special 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_bool 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 (([Init (Class C)], snd (fst s')),snd s')
))
       (xcpt_names (i,G,pc,et))"

  norm_eff :: "instr ⇒ jvm_prog ⇒ p_count ⇒ state_bool option ⇒ state_bool option"
  "norm_eff i G pc == option_map (eff_bool i G pc)"
```

Putting it all together:

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

Some small helpers for direct executability

```
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 × cname × p_count × nat × ty × state_type ⇒ bool"

recdef app' "{}"
"app' (Load idx, G, C', pc, maxs, rT, s)
  = (idx < length (snd s) ∧ (snd s) ! idx ≠ Err ∧ length (fst s) < maxs)"

"app' (Store idx, G, C', pc, maxs, rT, (ts#ST, LT))
  = (idx < length LT)"

"app' (LitPush v, G, C', pc, maxs, rT, s)
  = (length (fst s) < maxs ∧ typeof (λt. None) v ≠ None)"

"app' (Getfield F C, G, C', pc, maxs, rT, (oT#ST, LT))
  = (is_class G C ∧ field (G,C) F ≠ None ∧ fst (the (field (G,C) F)) = C ∧
     G ⊢ oT ⪯i Init (Class C))"

"app' (Putfield F C, G, C', 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 ⪯i Init (Class C) ∧ G ⊢ vT ⪯i Init (snd (the (field (G,C) F))))"

"app' (New C, G, C', pc, maxs, rT, s)
  = (is_class G C ∧ length (fst s) < maxs ∧ UnInit C pc ∉ set (fst s))"

"app' (Checkcast C, G, C', pc, maxs, rT, (Init (RefT rt)#ST,LT))
  = is_class G C"

"app' (Pop, G, C', pc, maxs, rT, (ts#ST,LT))              = True"
"app' (Dup, G, C', pc, maxs, rT, (ts#ST,LT))              = (1+length ST < maxs)"
"app' (Dup_x1, G, C', pc, maxs, rT, (ts1#ts2#ST,LT))      = (2+length ST < maxs)"
"app' (Dup_x2, G, C', pc, maxs, rT, (ts1#ts2#ts3#ST,LT)) = (3+length ST < maxs)"
"app' (Swap, G, C', pc, maxs, rT, (ts1#ts2#ST,LT))        = True"

"app' (IAdd, G, C', pc, maxs, rT, (t1#t2#ST,LT))
  = (t1 = Init (PrimT Integer) ∧ t1 = t2)"

"app' (Ifcmpeq b, G, C', pc, maxs, rT, (Init ts#Init ts'#ST,LT))
```

```
  = (0 ≤ int pc + b ∧ (isPrimT ts ⟶ ts' = ts) ∧ (isRefT ts ⟶ isRefT ts'))"

"app' (Goto b, G, C', pc, maxs, rT, s)            = (0 ≤ int pc + b)"
"app' (Return, G, C', pc, maxs, rT, (T#ST,LT))    = (G ⊢ T ⪯i Init rT)"
"app' (Throw, G, C', pc, maxs, rT, (Init T#ST,LT)) = isRefT T"

"app' (Invoke C mn fpTs, G, C', pc, maxs, rT, s) =
   (length fpTs < length (fst s) ∧ mn ≠ init ∧
   (let apTs = rev (take (length fpTs) (fst s));
        X    = hd (drop (length fpTs) (fst s))
    in  is_class G C ∧
        list_all2 (λaT fT. G ⊢ aT ⪯i (Init fT)) apTs fpTs ∧
        G ⊢ X ⪯i Init (Class C)) ∧
        method (G,C) (mn,fpTs) ≠ None)"

"app' (Invoke_special C mn fpTs, G, C', pc, maxs, rT, s) =
   (length fpTs < length (fst s) ∧ mn = init ∧
   (let apTs = rev (take (length fpTs) (fst s));
        X    = (fst s)!length fpTs
    in  is_class G C ∧
        list_all2 (λaT fT. G ⊢ aT ⪯i (Init fT)) apTs fpTs ∧
        (∃rT' b. method (G,C) (mn,fpTs) = Some (C,rT',b)) ∧
        ((∃pc. X = UnInit C pc) ∨ (X = PartInit C' ∧ G ⊢ C' ≺C1 C))))"
```

— *C'* is the current class, the constructor must be called on the
— superclass (if partly initialized) or on the exact class that is
— to be constructed (if not yet initialized at all).
— In JCVM *Invoke_special* may also call another constructor of the same
— class (*C = C' ∨ C = super C'*)

```
"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"
```

constdefs

```
  app :: "instr ⇒ jvm_prog ⇒ cname ⇒ p_count ⇒ nat ⇒ ty ⇒ bool ⇒
          exception_table ⇒ state_bool option ⇒ bool"
  "app i G C' pc maxs rT ini et s ≡ case s of None ⇒ True | Some t ⇒
  let (s,z) = t in
    xcpt_app i G pc et ∧
    app' (i,G,C',pc,maxs,rT,s) ∧
    (ini ∧ i = Return ⟶ z) ∧
    (∀C m p. i = Invoke_special C m p ∧ (fst s)!length p = PartInit C' ⟶ ¬z)"
```

```
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 C' maxs rT pc ini et None = True" by simp


lemma appLoad[simp]:
"app (Load idx) G C' pc maxs rT ini et (Some s) =
 (∃ST LT z. s = ((ST,LT),z) ∧ idx < length LT ∧ LT!idx ≠ Err ∧ length ST < maxs)"
  by (cases s, auto)

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

lemma appLitPush[simp]:
"app (LitPush v) G C' pc maxs rT ini et (Some s) = (∃ST LT z. s = ((ST,LT),z) ∧ length
ST < maxs ∧ typeof (λv. None) v ≠ None)"
  by (cases s, auto)

lemma appGetField[simp]:
"app (Getfield F C) G C' pc maxs rT ini et (Some s) =
 (∃ oT vT ST LT z. s = ((oT#ST, LT),z) ∧ is_class G C ∧
  field (G,C) F = Some (C,vT) ∧ G ⊢ oT ⪯i (Init (Class C)) ∧
  (∀x ∈ set (match G (Xcpt NullPointer) pc et). is_class G x))"
  by (cases s, cases "2 <length (fst (fst s))", auto dest!: 1 2)

lemma appPutField[simp]:
"app (Putfield F C) G C' pc maxs rT ini et (Some s) =
 (∃ vT vT' oT ST LT z. s = ((vT#oT#ST, LT),z) ∧ is_class G C ∧
  field (G,C) F = Some (C, vT') ∧
  G ⊢ oT ⪯i Init (Class C) ∧ G ⊢ vT ⪯i Init vT' ∧
  (∀x ∈ set (match G (Xcpt NullPointer) pc et). is_class G x))"
  by (cases s, cases "2 <length (fst (fst s))", auto dest!: 1 2)

lemma appNew[simp]:
"app (New C) G C' pc maxs rT ini et (Some s) =
 (∃ST LT z. s=((ST,LT),z) ∧
  is_class G C ∧ length ST < maxs ∧
   UnInit C pc ∉ set ST ∧
  (∀x ∈ set (match G (Xcpt OutOfMemory) pc et). is_class G x))"
  by (cases s, auto)

lemma appCheckcast[simp]:
"app (Checkcast C) G C' pc maxs rT ini et (Some s) =
 (∃rT ST LT z. s = ((Init (RefT rT)#ST,LT),z) ∧ is_class G C ∧
  (∀x ∈ set (match G (Xcpt ClassCast) pc et). is_class G x))"
proof -
```

```
  { fix t ST LT z assume "s = ((Init t#ST,LT),z)"
    hence ?thesis by (cases t, auto)
  } thus ?thesis
    by (cases s, cases "fst s", cases "fst (fst s)", simp,
        cases "hd (fst (fst s))", auto)
qed

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

lemma appDup[simp]:
"app Dup G C' pc maxs rT ini et (Some s) =
 (∃ts ST LT z. s = ((ts#ST,LT),z) ∧ 1+length ST < maxs)"
  by (cases s, cases "2 < length (fst (fst s))", auto dest: 1 2)

lemma appDup_x1[simp]:
"app Dup_x1 G C' pc maxs rT ini et (Some s) =
 (∃ts1 ts2 ST LT z. s = ((ts1#ts2#ST,LT),z) ∧ 2+length ST < maxs)"
  by (cases s, cases "2 < length (fst (fst s))", auto dest: 1 2)

lemma appDup_x2[simp]:
"app Dup_x2 G C' pc maxs rT ini et (Some s) =
 (∃ts1 ts2 ts3 ST LT z. s = ((ts1#ts2#ts3#ST,LT),z) ∧ 3+length ST < maxs)"
  by (cases s, cases "2 < length (fst (fst s))", auto dest: 1 2)

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

lemma appIAdd[simp]:
"app IAdd G C' pc maxs rT ini et (Some s) =
 (∃ ST LT z. s = ((Init (PrimT Integer)#Init (PrimT Integer)#ST,LT),z))"
  by (cases s, cases "2 < length (fst (fst s))", auto dest: 1 2)

lemma appIfcmpeq[simp]:
"app (Ifcmpeq b) G C' pc maxs rT ini et (Some s) =
 (∃ts1 ts2 ST LT z. s = ((Init ts1#Init ts2#ST,LT),z) ∧ 0 ≤ b + int pc ∧
 ((∃p. ts1 = PrimT p ∧ ts2 = PrimT p) ∨
  (∃r r'. ts1 = RefT r ∧ ts2 = RefT r')))"
  apply (cases s)
  apply (cases "fst s")

  apply (case_tac aa)
   apply simp
  apply (case_tac list)
   apply (case_tac ab, simp, simp, simp)
```

```
  apply (case_tac ab)
  apply auto
  apply (case_tac ac)
    defer
    apply simp
   apply simp
  apply (case_tac ty)
  apply auto
  done

lemma appReturn[simp]:
"app Return G C' pc maxs rT ini et (Some s) =
  (∃T ST LT z. s = ((T#ST,LT),z) ∧ (G ⊢ T ≼i Init rT) ∧ (ini ⟶ z))"
  by (cases s, cases "2 <length (fst (fst s))", auto dest: 1 2)

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

lemma appThrow[simp]:
"app Throw G C' pc maxs rT ini et (Some s) =
 (∃ST LT z r. s=((Init (RefT r)#ST,LT),z) ∧ (∀C ∈ set (match_any G pc et). is_class
G C))"
  apply (cases s)
  apply (cases "fst s")

  apply (case_tac aa)
   apply simp
  apply (case_tac ab)
  apply auto
  done

lemma appInvoke[simp]:
"app (Invoke C mn fpTs) G C' pc maxs rT ini et (Some s) =
 (∃apTs X ST LT mD' rT' b' z.
  s = (((rev apTs) @ (X # ST), LT), z) ∧ mn ≠ init ∧
  length apTs = length fpTs ∧ is_class G C ∧
  (∀(aT,fT)∈set(zip apTs fpTs). G ⊢ aT ≼i (Init fT)) ∧
  method (G,C) (mn,fpTs) = Some (mD', rT', b') ∧ (G ⊢ X ≼i Init (Class C)) ∧
  (∀C ∈ set (match_any G pc et). is_class G C))"
(is "?app s = ?P s")
proof -
  note list_all2_def[simp]
  { fix a b z
  have "?app ((a,b),z) ⟹ ?P ((a,b),z)"
  proof -
    assume app: "?app ((a,b),z)"
    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)
    then obtain apTs ST where
      "a = rev apTs @ ST ∧ length apTs = length fpTs ∧ 0 < length ST"
      by blast
    hence "a = rev apTs @ ST ∧ length apTs = length fpTs ∧ ST ≠ []"
      by blast
    then obtain X ST' where
      "a = rev apTs @ X # ST'" "length apTs = length fpTs"
      by (simp add: neq_Nil_conv) blast
    with app show ?thesis by clarsimp blast
  qed }
  moreover obtain a b z where "s = ((a,b),z)" by (cases s, cases "fst s", simp)
  ultimately have "?app s ⟹ ?P s" by (simp only:)
  moreover
  have "?P s ⟹ ?app s" by (clarsimp simp add: min_def)
  ultimately
  show ?thesis by (rule iffI)
qed

lemma appInvoke_special[simp]:
"app (Invoke_special C mn fpTs) G C' pc maxs rT ini et (Some s) =
 (∃apTs X ST LT rT' b' z.
  s = (((rev apTs) @ X # ST, LT), z) ∧ mn = init ∧
  length apTs = length fpTs ∧ is_class G C ∧
  (∀(aT,fT)∈set(zip apTs fpTs). G ⊢ aT ≼i (Init fT)) ∧
  method (G,C) (mn,fpTs) = Some (C, rT', b') ∧
  ((∃pc. X = UnInit C pc) ∨ (X = PartInit C' ∧ G ⊢ C' ≺C1 C ∧ ¬z)) ∧
  (∀C ∈ set (match_any G pc et). is_class G C))"
(is "?app s = ?P s")
proof -
  note list_all2_def [simp]
  { fix a b z
  have "?app ((a,b),z) ⟹ ?P ((a,b),z)"
  proof -
    assume app: "?app ((a,b),z)"
    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)
    then obtain apTs ST where
      "a = rev apTs @ ST ∧ length apTs = length fpTs ∧ 0 < length ST"
```

```
      by blast
    hence "a = rev apTs @ ST ∧ length apTs = length fpTs ∧ ST ≠ []"
      by blast
    then obtain X ST' where
      "a = rev apTs @ X # ST'" "length apTs = length fpTs"
      by (simp add: neq_Nil_conv) blast
    with app show ?thesis by (simp add: nth_append) blast
  qed }
  moreover obtain a b z where "s = ((a,b),z)" by (cases s, cases "fst s", simp)
  ultimately have "?app s ⟹ ?P s" by (simp only:)
  moreover
  have "?P s ⟹ ?app s" by (clarsimp simp add: nth_append min_def) blast
  ultimately
  show ?thesis by (rule iffI)
qed


lemma replace_map_OK:
  "replace (OK x) (OK y) (map OK l) = map OK (replace x y l)"
proof -
  have "inj OK" by (blast intro: datatype_injI)
  thus ?thesis by (rule replace_map)
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 more 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 InitPrimT_InitPrimT:
  "(G ⊢ xb ≼i Init (PrimT p)) = (xb = Init (PrimT p))"
  by (cases xb, auto elim: widen.elims simp add: subtype_def)

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 (clarsimp 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_set_conv_sup_loc [rule_format]:
"∀b. length a = length b ⟶
  (∀(x,y)∈set (zip a b). G ⊢ x ≼i Init y) =
  (G ⊢ (map OK a) <=l (map OK (map Init 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 replace_UnInit:
  "⟦ G ⊢ a <=l b; X = UnInit C pc ∨ X = PartInit D ⟧ ⟹
  G ⊢ (replace (OK X) v a) <=l (replace (OK X) v b)"
proof -
  assume X: "X = UnInit C pc ∨ X = PartInit D"
  assume "G ⊢ a <=l b"
  then obtain
    l: "length a = length b" and
    a: "∀i. i < length a ⟶ G ⊢ a!i <=o b!i"
    by (unfold sup_loc_def Listn.le_def lesub_def)
       (auto simp add: list_all2_conv_all_nth)

  { fix i assume "i < length (replace (OK X) v a)"

    hence i: "i < length a" by (simp add: replace_def)
    hence G: "G ⊢ a!i <=o b!i" by (simp add: a)
    from l i
    have i2: "i < length b" by simp

    have "G ⊢ (replace (OK X) v a)!i <=o (replace (OK X) v b)!i"
    proof (cases "a!i = OK X")
      case True
      with G i i2 X
      have "b!i = OK X ∨ b!i = Err"
        by (simp add: sup_ty_opt_def Err.le_def lesub_def init_le_def
                split: err.splits init_ty.split_asm)
      with True i i2
      show ?thesis
        by (auto simp add: replace_def)
           (simp add: sup_ty_opt_def Err.le_def)
    next
      case False
      with G i i2 X
      have "b!i ≠ OK X"
        by (auto simp add: sup_ty_opt_def Err.le_def lesub_def init_le_def
            split: init_ty.split_asm err.splits)
      with False i i2 G
      show ?thesis by (simp add: replace_def)
    qed
  }

  with l show ?thesis
    by (unfold sup_loc_def Listn.le_def lesub_def)
       (auto simp add: list_all2_conv_all_nth replace_def)
qed



lemma replace_mapOK_UnInit:
  "⟦ G ⊢ map OK ST <=l map OK ST'; X = UnInit C pc ∨ X = PartInit D ⟧ ⟹
  G ⊢ map OK (replace X v ST) <=l map OK (replace X v ST')"
```

```
proof -
  assume X: "X = UnInit C pc ∨ X = PartInit D"
  assume "G ⊢ map OK ST <=l map OK ST'"
  then obtain
    l: "length ST = length ST'" and
    a: "∀i. i < length ST ⟶ G ⊢ ST!i ≼i ST'!i"
    by (unfold sup_loc_def Listn.le_def lesub_def)
       (auto simp add: list_all2_conv_all_nth)

  { fix i assume "i < length (replace X v ST)"

    hence i: "i < length ST" by (simp add: replace_def)
    hence G: "G ⊢ ST!i ≼i ST'!i" by (simp add: a)
    from l i
    have i2: "i < length ST'" by simp

    have "G ⊢ (replace X v ST)!i ≼i (replace X v ST')!i"
    proof (cases "ST!i = X")
      case True
      with G i i2 X
      have "ST'!i = X"
        by (simp add: init_le_def split: init_ty.split_asm)
      with True i i2
      show ?thesis by (simp add: replace_def)
    next
      case False
      with G i i2 X
      have "ST'!i ≠ X"
        by (auto simp add: init_le_def split: init_ty.split_asm)
      with False i i2 G
      show ?thesis by (simp add: replace_def)
    qed
  }

  with l show ?thesis
    by (unfold sup_loc_def Listn.le_def lesub_def)
       (auto simp add: list_all2_conv_all_nth replace_def)
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])
      thus ?thesis
      proof (elim exE conjE)
        fix a b assume "ls = a @ b" "length a = n'"
        with s have "l # ls = (l#a) @ b ∧ length (l#a) = n" by simp
        thus ?thesis by blast
      qed
    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)
  thus ?thesis
  proof (elim exE conjE)
    fix r d assume x: "x = r@d" "length r = n"
    with n have "∃b c. d = b#c" by (simp add: neq_Nil_conv)
    thus ?thesis
    proof (elim exE conjE)
      fix b c assume "d = b#c"
      with x have "x = (rev (rev r)) @ b # c ∧ length (rev r) = n" by simp
      thus ?thesis by blast
    qed
  qed
qed

lemmas [iff] = not_Err_eq

lemma UnInit_set:
  "⟦ G ⊢ map OK ST <=l map OK ST'; UnInit C pc ∉ set ST' ⟧
  ⟹ UnInit C pc ∉ set ST"
proof
  assume "UnInit C pc ∈ set ST"
  then obtain x y where
    "ST = x @ UnInit C pc # y"
    by (clarsimp simp add: in_set_conv_decomp)
  hence l: "length x < length (map OK ST) ∧ ST!length x = UnInit C pc"
    by (simp add: nth_append)
  moreover
  assume G: "G ⊢ map OK ST <=l map OK ST'"
  moreover
  from G
  have lm: "length (map OK ST) = length (map OK ST')" by (rule sup_loc_length)
  ultimately
  obtain T where
    "UnInit C pc = ST'!length x"
    by clarify (drule sup_loc_nth, assumption, clarsimp simp add: nth_map)
  with G l lm [symmetric]
```

```
  have "UnInit C pc ∈ set ST'"
    by (auto simp add: set_conv_nth sup_loc_length)
  moreover
  assume "UnInit C pc ∉ set ST'"
  ultimately
  show False 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


lemma app_mono:
"⟦G ⊢ s <=' s'; app i G C pc m rT ini et s'⟧ ⟹ app i G C pc m rT ini et s"
proof -
  { fix s1 s2 z
    assume G:   "G ⊢ s2 <=s s1"
    assume app: "app i G C pc m rT ini et (Some (s1,z))"

    note [simp] = sup_loc_length sup_loc_length_map

    have "app i G C pc m rT ini et (Some (s2,z))"
    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 dest: UnInit_set)
    next
      case Getfield
      with app G show ?thesis
        by (cases s2, clarsimp simp add: sup_state_Cons2) (rule init_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 ≼i (Init (Class cname))" and
             vT: "G⊢ vT ≼i (Init b)" and
             xc: "Ball (set (match G (Xcpt 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' ≼i oT" and
             vT': "G⊢ vT' ≼i vT"
    by (cases s2, auto simp add: sup_state_Cons2)
  moreover
  from vT' vT
  have "G ⊢ vT' ≼i (Init b)" by (rule init_trans)
  moreover
  from oT' oT
  have "G⊢ oT' ≼i (Init (Class cname))" by (rule init_trans)
  ultimately
  show ?thesis by (cases s2, auto simp add: Putfield xc)
next
  case Checkcast
  with app G show ?thesis
    by (cases s2, auto simp add: init_le_Init2 sup_state_Cons2
                          intro!: widen_RefT2)
next
  case Return
  with app G show ?thesis
    by (cases s2, auto simp add: sup_state_Cons2)
       (rule init_trans, assumption+, rule init_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 InitPrimT_InitPrimT)
next
  case Goto with app show ?thesis by simp
next
  case Ifcmpeq
  with app G show ?thesis
    apply (cases s2, auto simp add: sup_state_Cons2 InitPrimT_InitPrimT)
    apply (auto simp add: init_le_Init2 widen_RefT2)
    done
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
    w:  "∀(x,y) ∈ set (zip apTs list). G ⊢ x ⪯i Init y" and
    m:  "method (G, cname) (mname, list) = Some (mD', rT', b')" and
    mn: "mname ≠ init" and
    C:  "G ⊢ X ⪯i Init (Class cname)" and
    x:  "∀C ∈ set (match_any G pc et). is_class G C"
    by simp blast

  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' ⪯i 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' ⪯i Init (Class cname)" by (blast intro: init_trans)

  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 (map Init list)"
    by (simp add: all_set_conv_sup_loc)
  finally have "G ⊢ map OK apTs' <=l map OK (map Init list)" .

  with l' have w': "∀(x,y) ∈ set (zip apTs' list). G ⊢ x ⪯i Init y"
```

```
      by (simp add: all_set_conv_sup_loc)

    from Invoke s2 l' w' C' m c mn x
    show ?thesis by (simp del: split_paired_Ex) blast
  next
    case Invoke_special

    with app
    obtain apTs X ST LT rT' b' where
      s1: "s1 = (rev apTs @ X # ST, LT)" and
      l:  "length apTs = length list" and
      c:  "is_class G cname" and
      w:  "∀(x,y) ∈ set (zip apTs list). G ⊢ x ⪯i Init y" and
      m:  "method (G, cname) (mname, list) = Some (cname, rT', b')" and
      mn: "mname = init" and
      C:  "(∃pc. X = UnInit cname pc) ∨ (X = PartInit C ∧ G ⊢ C ≺C1 cname ∧ ¬ z)"
and
      x:  "∀C ∈ set (match_any G pc et). is_class G C"
      by simp blast

    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' ⪯i 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':
      "(∃pc. X' = UnInit cname pc) ∨ (X' = PartInit C ∧ G ⊢ C ≺C1 cname ∧ ¬ z)"
      by 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 (map Init list)"
      by (simp add: all_set_conv_sup_loc)
    finally have "G ⊢ map OK apTs' <=l map OK (map Init list)" .

    with l' have w': "∀(x,y) ∈ set (zip apTs' list). G ⊢ x ⪯i Init y"
      by (simp add: all_set_conv_sup_loc)

    from Invoke_special s2 l' w' C' m c mn 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 init_le_Init2 widen_RefT2)
    qed
  } note this [simp]

  assume "G ⊢ s <=' s'" "app i G C pc m rT ini et s'"
  thus ?thesis  by (cases s, cases s', auto)
qed


lemmas [simp del] = split_paired_Ex


lemma eff_bool_mono:
"⟦ app i G C pc m rT ini et (Some t); G ⊢ s <=b t ⟧ ⟹
  G ⊢ eff_bool i G pc s <=b eff_bool i G pc t"
proof -
  obtain s1 z where s': "s = (s1,z)" by (cases s)
  moreover assume "G ⊢ s <=b t"
  ultimately
  obtain s2 where t: "t = (s2,z)" and G: "G ⊢ s1 <=s s2" by (cases t, simp)

  from s' t obtain a1 b1 a2 b2 where s12: "s1 = (a1,b1)" "s2 = (a2,b2)"
    by (cases s1, cases s2)
  moreover assume "app i G C pc m rT ini et (Some t)"
  moreover note t
  ultimately
  have app2: "app i G C pc m rT ini et (Some (s2,z))" by simp

  have "G ⊢ (Some (s1,z)) <=' (Some (s2,z))" by simp
  from this app2
  have app1: "app i G C pc m rT ini et (Some (s1,z))" by (rule app_mono)

  note [simp] = eff_def eff_bool_def
  note s = s' t s12

  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 ≼i 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 sup_state_conv)
       (blast intro: replace_UnInit)
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 = (rev a @ X # ST, b1)" and
    l:  "length a = length list" and
    C:  "G ⊢ X ≼i Init (Class cname)"
    by (clarsimp, blast)

  from Invoke s app2 obtain a' X' ST' where
    s2: "s2 = (rev a' @ X' # ST', b2)" and
    l': "length a' = length list" and
    C': "G ⊢ X' ≼i Init (Class cname)"
    by (clarsimp, blast)

  from l l' have "length a = length a'" by simp
  from this G s1 s2 have "G ⊢ (ST, b1) <=s (ST', b2)"
    by (simp add: sup_state_append_fst sup_state_Cons1)

  with Invoke G app1 app2 s s1 s2 l l'
  show ?thesis by (clarsimp simp add: sup_state_conv)
next
  case Return
  with G s 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 Invoke_special

    with s app1
    obtain a X ST where
      s1: "s1 = (rev a @ X # ST, b1)" and
      l:  "length a = length list" and
      C:  "(∃pc. X = UnInit cname pc) ∨ X = PartInit C ∧ G ⊢ C ≺C1 cname ∧ ¬ z"
      by (clarsimp, blast)

    from Invoke_special s app2
    obtain a' X' ST' where
      s2: "s2 = (rev a' @ X' # ST', b2)" and
      l': "length a' = length list" and
      C': "(∃pc. X' = UnInit cname pc) ∨ X' = PartInit C ∧ G ⊢ C ≺C1 cname ∧ ¬ z"
      by (clarsimp, blast)

    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
    from lr G s1 s2
    have "G ⊢ X ⪯i X'" by (simp add: sup_state_append_fst sup_state_Cons1)
    with C C' have XX': "X = X'" by auto
    moreover
    note Invoke_special G app1 app2 s s1 s2 l l' C C'
    ultimately show ?thesis
      by (clarsimp simp add: sup_state_conv nth_append)
         (blast intro: replace_UnInit replace_mapOK_UnInit)
  next
    case Throw with G s 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,cname,ty,method_type,nat,bool,
              p_count,exception_table,p_count] ⇒ bool"
"wt_instr i G C rT phi mxs ini max_pc et pc ≡
  app i G C pc mxs rT ini 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,mname,ty list,nat,method_type] ⇒ bool"
"wt_start G C mn pTs mxl phi ≡
let
  this  = OK (if mn = init ∧ C ≠ Object then PartInit C else Init (Class C));
  start = Some (([],this#(map OK (map Init pTs))@(replicate mxl Err)),C=Object)
in
  G ⊢ start <=' 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,mname,ty list,ty,nat,nat,
               instr list,exception_table,method_type] ⇒ bool"
"wt_method G C mn 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 mn pTs mxl phi ∧
      (∀pc. pc<max_pc ⟶ wt_instr (ins!pc) G C rT phi mxs (mn=init) max_pc et pc)"
```

```
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 (fst sig) (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 C rT (phi C sig) maxs (fst sig=init) (length ins) et pc"
by (unfold wt_jvm_prog_def, drule method_wf_mdecl,
    simp, simp, simp add: wf_mdecl_def wt_method_def)
```

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 (fst sig) (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)
```

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; ini = (fst sig=init) ⟧
  ⟹ wt_instr i G C rT phi maxs ini max_pc et pc =
     (app i G C pc maxs rT ini 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

end
```

4.15 The Typing Framework for the JVM

theory *Typing_Framework_JVM = Typing_Framework_err + JVMType + EffectMono + BVSpec:*

constdefs
```
  exec :: "jvm_prog ⇒ cname ⇒ nat ⇒ ty ⇒ bool ⇒ exception_table ⇒ instr list ⇒

           state step_type"
  "exec G C maxs rT ini et bs ==
  err_step (size bs) (λpc. app (bs!pc) G C pc maxs rT ini et) (λpc. eff (bs!pc) G pc et)"
```

constdefs
```
  opt_states :: "'c prog ⇒ nat ⇒ nat ⇒ ((init_ty list × init_ty err list) × bool)
option set"
  "opt_states G maxs maxr ≡ opt ((⋃{list n (init_tys G) |n. n ≤ maxs} × list maxr (err
(init_tys G))) × {True,False})"
```

locale *JVM_sl =*
  fixes *wf_mb* **and** *G* **and** *C* **and** *mxs* **and** *mxl*
  fixes *pTs :: "ty list"* **and** *mn* **and** *bs* **and** *et* **and** *rT*

  fixes *mxr* **and** *A* **and** *r* **and** *app* **and** *eff* **and** *step*
  defines *[simp]: "mxr ≡ 1+length pTs+mxl"*
  defines *[simp]: "A   ≡ states G mxs mxr"*
  defines *[simp]: "r   ≡ JVMType.le G mxs mxr"*

  defines *[simp]: "app ≡ λpc. Effect.app (bs!pc) G C pc mxs rT (mn=init) et"*
  defines *[simp]: "eff ≡ λpc. Effect.eff (bs!pc) G pc et"*
  defines *[simp]: "step ≡ exec G C mxs rT (mn=init) et bs"*

locale (open) *start_context = JVM_sl +*
  assumes *wf: "wf_prog wf_mb G"*
  assumes *C: "is_class G C"*
  assumes *pTs: "set pTs ⊆ types G"*

  fixes *this* **and** *first :: "state_bool option"* **and** *start*
  defines *[simp]:*
  "this ≡ OK (if mn=init ∧ C ≠ Object then PartInit C else Init (Class C))"
  defines *[simp]:*
  "first ≡ Some (([],this#(map (OK∘Init) pTs)@(replicate mxl Err)), C=Object)"
  defines *[simp]:*
  "start ≡ OK first#(replicate (size bs - 1) (OK None))"

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)"
  apply (unfold bounded_def eff_def)
  apply clarify
  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 *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*

```
lemmas [simp] = init_tys_def JType.esl_def

lemma replace_in_setI:
  "⋀n. ls ∈ list n A ⟹ b ∈ A ⟹ replace a b ls ∈ list n A"
  by (induct ls) (auto simp add: replace_def)

theorem exec_pres_type:
  "wf_prog wf_mb S ⟹
  pres_type (exec S C maxs rT ini 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 eff_bool_def xcpt_eff_def norm_eff_def)
  apply (case_tac "bs!p")

  — load
  apply (clarsimp simp add: not_Err_eq)
  apply (drule listE_nth_in, assumption)
  apply fastsimp

  — store
  apply (fastsimp simp add: not_None_eq)

  — litpush
  apply clarsimp
  apply (rule_tac x="Suc n" in exI)
  apply (fastsimp simp add: not_None_eq typeof_empty_is_type)

  — new
  apply clarsimp
  apply (erule disjE)
   apply (fastsimp intro: replace_in_setI)
  apply clarsimp
  apply (rule_tac x=1 in exI)
  apply fastsimp

  — getfield
  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

  — putfield
  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
```

— checkcast

```
apply clarsimp
apply (erule disjE)
 apply fastsimp
apply clarsimp
apply (rule_tac x=1 in exI)
apply fastsimp
```

— invoke

```
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
```

— *invoke_special*

```
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 intro: replace_in_setI subcls_is_class)
apply clarsimp
apply (rule_tac x=1 in exI)
apply fastsimp
```

— return

```
apply fastsimp
```

— pop

```
apply fastsimp
```

— dup

```
apply clarsimp
apply (rule_tac x="n'+2" in exI)
apply fastsimp
```

— *dup_x1*

```
apply clarsimp
apply (rule_tac x="Suc (Suc (Suc (length ST)))" in exI)
apply fastsimp
```

— *dup_x2*

```
apply clarsimp
apply (rule_tac x="Suc (Suc (Suc (Suc (length ST))))" in exI)
apply fastsimp
```

— swap

```
apply fastsimp
```

```
  — iadd
  apply fastsimp

  — goto
  apply fastsimp

  — icmpeq
  apply fastsimp

  — throw
  apply clarsimp
  apply (erule disjE)
   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 (Product.le (Listn.le (init_le G)) (Listn.le (Err.le
(init_le G)))) (op =))"
  by (simp add: sup_state_opt_def sup_state_bool_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 C pc maxs rT ini et) (length bs) (opt_states
G maxs maxr)"
  by (unfold app_mono_def lesub_def) (blast intro: EffectMono.app_mono)

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 C p maxs rT ini 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_bool_mono, 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 intro: order_init eq_order)


theorem exec_mono:
  "wf_prog wf_mb G ⟹ bounded (exec G C maxs rT ini et bs) (size bs) ⟹
  mono (JVMType.le G maxs maxr) (exec G C maxs rT ini 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_Product_esl)
```

```
  apply (rule err_semilat_upto_esl)
  apply (rule err_semilat_init, assumption)
  apply (rule err_semilat_eslI)
  apply (rule Listn_sl)
  apply (rule err_semilat_init, assumption)
  apply (rule bool_err_semilat)
  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 (in JVM_sl) wt_method_def2:
  "wt_method G C mn 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 mn pTs mxl phi ∧
   wt_app_eff (sup_state_opt G) app eff phi)"
  by (auto simp add: wt_method_def wt_app_eff_def wt_instr_def lesub_def
           dest: check_bounded_is_bounded boundedD)


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'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
```

4.17 Kildall for the JVM

```
theory JVM = Typing_Framework_JVM + Kildall:

constdefs
  kiljvm :: "jvm_prog ⇒ cname ⇒ nat ⇒ nat ⇒ ty ⇒ bool ⇒ exception_table ⇒
             instr list ⇒ state list ⇒ state list"
  "kiljvm G C maxs maxr rT ini et bs ==
  kildall (JVMType.le G maxs maxr) (JVMType.sup G maxs maxr) (exec G C maxs rT ini et
bs)"


  wt_kil :: "jvm_prog ⇒ cname ⇒ ty list ⇒ ty ⇒ bool ⇒ nat ⇒ nat
             ⇒ exception_table ⇒ instr list ⇒ bool"
  "wt_kil G C pTs rT ini mxs mxl et ins ==
  check_bounded ins et ∧ 0 < size ins ∧
  (let this   = OK (if ini ∧ C ≠ Object then PartInit C else Init (Class C));
       first  = Some (([],this#((map OK (map Init pTs)))@(replicate mxl Err)),
                      C=Object);
       start  = OK first#(replicate (size ins - 1) (OK None));
       result = kiljvm G C mxs (1+size pTs+mxl) rT ini 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 (fst sig=init) maxs maxl et b) G"



theorem (in start_context) is_bcv_kiljvm:
  "bounded step (size bs) ⟹
  is_bcv r Err step (size bs) A (kiljvm G C mxs mxr rT (mn=init) et bs)"
  apply simp
  apply (insert wf)
  apply (unfold kiljvm_def)
  apply (rule is_bcv_kildall)
       apply (simp (no_asm) add: sl_triple_conv [symmetric])
       apply (rule semilat_JVM_slI, assumption)
      apply (simp (no_asm) add: JVM_le_unfold)
      apply (blast intro!: ac_order_init wf_converse_subcls1_impl_acc_init
                   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

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 (in start_context) start_in_A [intro?]:
  "0 < size bs ⟹ start ∈ list (size bs) A"
    apply (insert pTs C)
    apply (simp split del: split_if)
    apply (unfold JVM_states_unfold)
    apply (auto simp add: map_compose split
                intro!: listI list_appendI dest!: in_set_replicate)
    apply force+
    done

theorem (in start_context) wt_kil_correct:
  assumes wtk: "wt_kil G C pTs rT (mn=init) mxs mxl et bs"
  shows "∃phi. wt_method G C mn pTs rT mxs mxl bs et phi"
proof -
  from wtk obtain res where
    ck_bound: "check_bounded bs et" and
    result:   "res = kiljvm G C mxs mxr rT (mn=init) et bs start" and
    success:  "∀n < size bs. res!n ≠ Err" and
    instrs:   "0 < size bs"
    by (unfold wt_kil_def) (simp add: map_compose)

  from ck_bound have bounded: "bounded step (size bs)"
    by (simp add: exec_def) (intro bounded_lift check_bounded_is_bounded)
  hence bcv:
    "is_bcv r Err step (size bs) A (kiljvm G C mxs mxr rT (mn=init) et bs)"
    by (rule is_bcv_kiljvm)

  from instrs have "start ∈ list (size bs) A" ..
  with bcv success result have
    "∃ts∈list (size bs) A. start <=[r] ts ∧ wt_step r Err step ts"
    by (unfold is_bcv_def) blast
  then obtain phi' where
    in_A: "phi' ∈ list (size bs) A" and
    s:    "start <=[r] phi'" and
    w:    "wt_step r Err step phi'"
    by blast
  hence wt_err_step: "wt_err_step (sup_state_opt G) step phi'"
    by (simp add: wt_err_step_def JVM_le_Err_conv)

  from in_A have l: "size phi' = size bs" by simp
```

```
  moreover {
    from in_A  have "check_types G mxs mxr phi'" by (simp add: check_types_def)
    also from w
    have [symmetric]: "map OK (map ok_val phi') = phi'"
      by (auto intro!: map_id simp add: wt_step_def not_Err_eq)
    finally  have "check_types G mxs mxr (map OK (map ok_val phi'))" .
  }
  moreover {
    from s have "start!0 <=_r phi'!0" by (rule le_listD) simp
    moreover
    from instrs w l
    have "phi'!0 ≠ Err" by (unfold wt_step_def) simp
    then obtain phi0 where "phi'!0 = OK phi0" by (auto simp add: not_Err_eq)
    ultimately
    have "wt_start G C mn pTs mxl (map ok_val phi')" using l instrs
      by (unfold wt_start_def) (simp add: map_compose lesub_def JVM_le_Err_conv)
  }
  moreover
  from bounded wt_err_step
  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: l exec_def)
  ultimately
  have "wt_method G C mn pTs rT mxs mxl bs et (map ok_val phi')"
    using instrs ck_bound by (simp add: wt_method_def2)
  thus ?thesis by blast
qed


theorem (in start_context) wt_kil_complete:
  assumes wtm: "wt_method G C mn pTs rT mxs mxl bs et phi"
  shows "wt_kil G C pTs rT (mn=init) mxs mxl et bs"
proof -
  from wtm obtain
    instrs:   "0 < size bs" and
    length:   "length phi = length bs" and
    ck_bound: "check_bounded bs et" and
    ck_type:  "check_types G mxs mxr (map OK phi)" and
    wt_start: "wt_start G C mn pTs mxl phi" and
    app_eff:  "wt_app_eff (sup_state_opt G) app eff phi"
    by (simp add: wt_method_def2)

  from ck_bound have bounded: "bounded step (size bs)"
    by (simp add: exec_def) (intro bounded_lift check_bounded_is_bounded)
  with app_eff
  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 length)
  hence wt_err: "wt_err_step (sup_state_opt G) step (map OK phi)"
    by (simp add: exec_def length)

  from bounded have is_bcv:
    "is_bcv r Err step (size bs) A (kiljvm G C mxs mxr rT (mn=init) et bs)"
    by (rule is_bcv_kiljvm)
  moreover
  from instrs have "start ∈ list (size bs) A" ..
```

```
  moreover
  let ?phi = "map OK phi"
  have less_phi: "start <=[r] ?phi"
  proof (rule le_listI)
    from length instrs
    show "length start = length (map OK phi)" by simp
  next
    fix n
    from wt_start have "G ⊢ ok_val (start!0) <=' phi!0"
      by (simp add: wt_start_def map_compose)
    moreover from instrs length have "0 < length phi" by simp
    ultimately have "start!0 <=_r ?phi!0"
      by (simp add: JVM_le_Err_conv Err.le_def lesub_def)
    moreover {
      fix n'
      have "OK None <=_r ?phi!n"
        by (auto simp add: JVM_le_Err_conv Err.le_def lesub_def
                 split: err.splits)
      hence "⟦ n = Suc n'; n < size start ⟧
             ⟹ start!n <=_r ?phi!n" by simp
    }
    ultimately
    show "n < size start ⟹ start!n <=_r ?phi!n" by (cases n, blast+)
  qed
  moreover
  from ck_type length
  have "?phi ∈ list (size bs) A"
    by (auto intro!: listI simp add: check_types_def)
  moreover
  from wt_err have "wt_step r Err step ?phi"
    by (simp add: wt_err_step_def JVM_le_Err_conv)
  ultimately
  have "∀p. p < size bs ⟶ kiljvm G C mxs mxr rT (mn=init) et bs start ! p ≠ Err"
    by (unfold is_bcv_def) blast
  with ck_bound instrs
  show "wt_kil G C pTs rT (mn=init) mxs mxl et bs"
    by (unfold wt_kil_def) (simp add: map_compose)
qed


theorem jvm_kildall_correct:
  "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 (fst sig) (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 simp
    apply (auto dest!: start_context.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: start_context.wt_kil_complete)
    done
qed


end
```

4.18 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]
```

4.18.1 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)
```

4.18.2 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
```

4.18.3 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
```

4.18.4 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
```

4.19 Correctness of the LBV

```
theory LBVCorrect = LBVSpec + Typing_Framework:

locale (open) lbvs = lbv +
  fixes s0  :: 'a ("s₀")
  fixes c   :: "'a list"
  fixes ins :: "'b list"
  fixes phi :: "'a list" ("φ")
  defines phi_def:
  "φ ≡ map (λpc. if c!pc = ⊥ then wtl (take pc ins) c 0 s0 else c!pc)
       [0..length ins(]"

  assumes bounded: "bounded step (length ins)"
  assumes cert: "cert_ok c (length ins) ⊤ ⊥ A"
  assumes pres: "pres_type step (length ins) A"


lemma (in lbvs) phi_None [intro?]:
  "⟦ pc < length ins; c!pc = ⊥ ⟧ ⟹ φ ! pc = wtl (take pc ins) c 0 s0"
  by (simp add: phi_def)

lemma (in lbvs) phi_Some [intro?]:
  "⟦ pc < length ins; c!pc ≠ ⊥ ⟧ ⟹ 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 s₀ ≠ ⊤"
  assumes pc:  "pc+1 < length ins"
  shows "wtl (take (pc+1) ins) c 0 s0 ≤ᵣ φ!(pc+1)"
proof -
  from all pc
  have "wtc c (pc+1) (wtl (take (pc+1) ins) c 0 s0) ≠ 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 ≠ ⊤"
  assumes s0:  "s0 ∈ A"
  assumes pc:  "pc < length ins"
  shows "stable r step φ pc"
proof (unfold stable_def, clarify)
  fix pc' s' assume step: "(pc',s') ∈ set (step pc (φ ! pc))"
                      (is "(pc',s') ∈ set (?step pc)")

  from bounded pc step have pc': "pc' < length ins" by (rule boundedD)

  have tkpc: "wtl (take pc ins) c 0 s0 ≠ ⊤" (is "?s1 ≠ _") 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
```

4.20 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

4.21 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 ⇒ cname ⇒ nat ⇒ nat ⇒ ty ⇒ bool ⇒ exception_table ⇒
            state list ⇒ instr list ⇒ state ⇒ state"
  "lbvjvm G C maxs maxr rT ini et cert bs ≡
  wtl_inst_list bs cert (JVMType.sup G maxs maxr) (JVMType.le G maxs maxr) Err (OK None)
(exec G C maxs rT ini et bs) 0"

  wt_lbv :: "jvm_prog ⇒ cname ⇒ mname ⇒ ty list ⇒ ty ⇒ nat ⇒ nat ⇒
            exception_table ⇒ state list ⇒ instr list ⇒ bool"
  "wt_lbv G C mn 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 this  = OK (if mn=init ∧ C ≠ Object then PartInit C else Init (Class C));
        start = Some (([],this#(map (OK∘Init) pTs)@(replicate mxl Err)),C=Object);
        result = lbvjvm G C mxs (1+size pTs+mxl) rT (mn=init) 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 (fst sig) (snd sig) rT maxs maxl
et (cert C sig) b) G"

  mk_cert :: "jvm_prog ⇒ cname ⇒ nat ⇒ ty ⇒ bool ⇒ exception_table ⇒ instr list

             ⇒ method_type ⇒ state list"
  "mk_cert G C maxs rT ini et bs phi ≡ make_cert (exec G C maxs rT ini 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 C maxs rT (fst sig=init) et ins (phi C sig)"


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 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 [intro]:
  "x ∈ A ⟹ replicate n x ∈ list n A"
  by (induct n, auto)

lemma (in start_context) first_in_A: "OK first ∈ A"
  apply (insert pTs C)
  apply (simp add: JVM_states_unfold)
  apply (auto intro!: list_appendI)
  apply force+
  done

lemma (in start_context) wt_lbv_wt_step:
  assumes lbv: "wt_lbv G C mn pTs rT mxs mxl et cert bs"
  defines [simp]: "f ≡ JVMType.sup G mxs mxr"
  shows "∃ts ∈ list (size bs) A. wt_step r Err step ts ∧ OK first <=_r ts!0"
proof -
  have "semilat (JVMType.sl G mxs mxr)" by (rule semilat_JVM_slI)
  hence "semilat (A, r, f)" by (simp add: 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 bs)"
    by (clarsimp simp add: wt_lbv_def exec_def)
       (intro bounded_lift check_bounded_is_bounded)
  moreover
  from lbv
  have "cert_ok cert (length bs) Err (OK None) A"
    by (unfold wt_lbv_def) (auto dest: check_certD)
  moreover
  have "pres_type step (length bs) A" by simp (rule exec_pres_type)
  moreover
  from lbv
  have "wtl_inst_list bs cert f r Err (OK None) step 0 (OK first) ≠ Err"
    by (simp add: wt_lbv_def lbvjvm_def)
  moreover
  note first_in_A
  moreover
```

```
  from lbv have "0 < length bs" by (simp add: wt_lbv_def)
  ultimately
  show ?thesis by (rule lbvs.wtl_sound_strong)
qed


lemma (in start_context) wt_lbv_wt_method:
  assumes lbv: "wt_lbv G C mn pTs rT mxs mxl et cert bs"

  shows "∃phi. wt_method G C mn pTs rT mxs mxl bs et phi"
proof -
  from lbv have l: "bs ≠ []" by (simp add: wt_lbv_def)
  moreover
  from wf lbv C pTs
  obtain phi where
    list:  "phi ∈ list (length bs) A" and
    step:  "wt_step r Err step phi" and
    start: "OK first <=_r phi!0"
    by (blast dest: wt_lbv_wt_step)
  from list have [simp]: "length phi = length bs" by simp
  have "length (map ok_val phi) = length bs" 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 mn pTs mxl (map ok_val phi)"
    by (simp add: wt_start_def JVM_le_Err_conv lesub_def map_compose)
  moreover
  from lbv  have chk_bounded: "check_bounded bs 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 {
    from chk_bounded
    have "bounded (err_step (length bs) app eff) (length bs)"
      by simp (blast intro: bounded_lift check_bounded_is_bounded)
    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 mn pTs rT mxs mxl bs et (map ok_val phi)"
```

```
    by (simp add: wt_method_def2)
  thus ?thesis ..
qed


lemma (in start_context) wt_method_wt_lbv:
  assumes wt: "wt_method G C mn pTs rT mxs mxl bs et phi"

  defines [simp]: "cert ≡ mk_cert G C mxs rT (mn=init) et bs phi"
  defines [simp]: "f ≡ JVMType.sup G mxs mxr"

  shows "wt_lbv G C mn pTs rT mxs mxl et cert bs"
proof -
  let ?phi  = "map OK phi"
  let ?cert = "make_cert step ?phi (OK None)"

  from wt obtain
    0:          "0 < length bs" and
    length:     "length bs = length ?phi" and
    ck_bounded: "check_bounded bs et" and
    ck_types:   "check_types G mxs mxr ?phi" and
    wt_start:   "wt_start G C mn pTs mxl phi" and
    app_eff:    "wt_app_eff (sup_state_opt G) app eff phi"
    by (force simp add: wt_method_def2)

  have "semilat (JVMType.sl G mxs mxr)" by (rule semilat_JVM_slI)
  hence "semilat (A, r, f)" by (simp add: 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 bs)"
    by (clarsimp simp add: exec_def)
       (intro bounded_lift check_bounded_is_bounded)
  with wf
  have "mono r step (length bs) A" by simp (rule exec_mono)
  hence "mono r step (length ?phi) A" by (simp add: length)
  moreover
  have "pres_type step (length bs) A" by simp (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 step (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
  from 0 length have "0 < length phi" by auto
  hence "?phi!0 = OK (phi!0)" by simp
  with wt_start have "OK first <=_r ?phi!0"
    by (clarsimp simp add: wt_start_def lesub_def JVM_le_Err_conv map_compose)
  moreover
  note first_in_A
  moreover
  have "OK first ≠ Err" by simp
  moreover
  note length
  ultimately
  have "wtl_inst_list bs ?cert f r Err (OK None) step 0 (OK first) ≠ 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,bs,et)) = the (method (G,C) sig) in
              SOME phi. wt_method G C (fst sig) (snd sig) rT maxs maxl bs 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: start_context.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 start_context.wt_method_wt_lbv)*
done

end

4.22 BV Type Safety Invariant

theory *Correct = BVSpec + JVMExec:*

The fields of all objects contain only fully initialized objects:

constdefs

```
l_init :: "'c prog ⇒ aheap ⇒ init_heap ⇒ ('a ⇝ val) ⇒ ('a ⇝ ty) ⇒ bool"
"l_init G hp ih vs Ts ==
∀n T. Ts n = Some T ⟶ (∃v. vs n = Some v ∧ is_init hp ih v)"

o_init :: "'c prog ⇒ aheap ⇒ init_heap ⇒ obj ⇒ bool"
"o_init G hp ih obj == l_init G hp ih (snd obj) (map_of (fields (G,fst obj)))"

h_init :: "'c prog ⇒ aheap ⇒ init_heap ⇒ bool"
"h_init G h ih == ∀a obj. h a = Some obj ⟶ o_init G h ih obj"
```

Type and init information conforms. For uninitialized objects dynamic+static type must be the same

constdefs

```
iconf :: "'c prog ⇒ aheap ⇒ init_heap ⇒ val ⇒ init_ty ⇒ bool"
         ("_,_,_ ⊢ _ ::≼i _" [51,51,51,51,51] 50)
"G,h,ih ⊢ v ::≼i T ==
case T of Init ty ⇒ G,h⊢v::≼ty ∧ is_init h ih v
        | UnInit C pc ⇒  G,h⊢v::≼(Class C) ∧
                         (∃l fs. v = Addr l ∧ h l = Some (C,fs) ∧ ih l = T)
        | PartInit C ⇒ G,h⊢v::≼(Class C) ∧
                       (∃l. v = Addr l ∧ h l ≠ None ∧ ih l = T)"
```

Alias analysis for uninitialized objects is correct. If two values have the same uninitialized type, they must be equal and marked with this type on the type tag heap.

constdefs

```
corr_val :: "[val, init_ty, init_heap] ⇒ bool"
"corr_val v T ihp == ∃l. v = Addr l ∧ ihp l = T"

corr_loc :: "[val list, locvars_type, init_heap, val, init_ty] ⇒ bool"
"corr_loc loc LT ihp v T ==
list_all2 (λl t. t = OK T ⟶ l = v ∧ corr_val v T ihp) loc LT"

corr_stk :: "[opstack, opstack_type, init_heap, val, init_ty] ⇒ bool"
"corr_stk stk ST ihp v T == corr_loc stk (map OK ST) ihp v T"

corresponds :: "[opstack,val list,state_type,init_heap,val,init_ty] ⇒ bool"
"corresponds stk loc s ihp v T ==
corr_stk stk (fst s) ihp v T ∧ corr_loc loc (snd s) ihp v T"

consistent_init :: "[opstack, val list, state_type, init_heap] ⇒ bool"
"consistent_init stk loc s ihp ==
(∀C pc. ∃v. corresponds stk loc s ihp v (UnInit C pc)) ∧
(∀C. ∃v. corresponds stk loc s ihp v (PartInit C))"
```

The values on stack and local variables conform to their static type:

constdefs

```
approx_val :: "[jvm_prog,aheap,init_heap,val,init_ty err] ⇒ bool"
"approx_val G h i v any == case any of Err ⇒ True | OK T ⇒ G,h,i ⊢ v ::≼i T"

approx_loc :: "[jvm_prog,aheap,init_heap,val list,locvars_type] ⇒ bool"
"approx_loc G hp i loc LT == list_all2 (approx_val G hp i) loc LT"

approx_stk :: "[jvm_prog,aheap,init_heap,opstack,opstack_type] ⇒ bool"
"approx_stk G hp i stk ST == approx_loc G hp i stk (map OK ST)"
```

A call frame is correct, if stack and local variables conform, if the types UnInit and PartInit are used consistently, if the `this` pointer in constructors is tagged correctly and its type is only used for the `this` pointer, if the *pc* is inside the method, and if the number of local variables is correct.

constdefs

```
correct_frame  :: "[jvm_prog,aheap,init_heap,state_type,nat,bytecode]
                   ⇒ frame ⇒ bool"
"correct_frame G hp i == λ(ST,LT) maxl ins (stk,loc,C,sig,pc,r).
approx_stk G hp i stk ST  ∧
approx_loc G hp i loc LT ∧
consistent_init stk loc (ST,LT) i ∧
(fst sig = init ⟶
 corresponds stk loc (ST,LT) i (fst r) (PartInit C) ∧
 (∃l. fst r = Addr l ∧ hp l ≠ None ∧
      (i l = PartInit C ∨ (∃C'. i l = Init (Class C'))))) ∧
pc < length ins ∧
length loc=length(snd sig)+maxl+1"
```

The reference update between constructors must be correct. In this predicate,

a is the this pointer of the calling constructor (the reference to be initialized), it must be of type UnInit or PartInit.

b is the this pointer of the current constructor, it must be partly initialized up to the current class.

c is the fully initialized object it can be null (before super is called), or must be a fully initialized object with type of the original (calling) class (z from the BV is true iff c contains an object)

C is the class where the current constructor is declared

constdefs

```
constructor_ok :: "[jvm_prog,aheap,init_heap,val,cname,bool,ref_upd] ⇒ bool"
"constructor_ok G hp ih a C z == λ(b,c).
z = (c ≠ Null) ∧ (∃C' D pc l l' fs1 fs2. a = Addr l ∧ b = Addr l' ∧
(ih l = UnInit C' pc ∨ ih l = PartInit D) ∧ hp l = Some (C',fs1) ∧
(ih l' = PartInit C ∨ ih l' = Init (Class C')) ∧ hp l' = Some (C',fs2) ∧
```

```
  (c ≠ Null ⟶
  (∃loc. c = Addr loc ∧ (∃fs3. hp loc = Some (C',fs3)) ∧ ih loc = Init (Class C'))))"
```

The whole call frame stack must be correct (the topmost frame is handled separately)

consts

```
  correct_frames  :: "[jvm_prog,aheap,init_heap,prog_type,ty,sig,
                       bool,ref_upd,frame list] ⇒ bool"
```

primrec

```
  "correct_frames G hp i phi rT0 sig0 z r [] = True"

  "correct_frames G hp i phi rT0 sig0 z0 r0 (f#frs) =
        (let (stk,loc,C,sig,pc,r) = f in
  (∃ST LT z rT maxs maxl ins et.
    phi C sig ! pc = Some ((ST,LT),z) ∧ is_class G C ∧
    method (G,C) sig = Some(C,rT,(maxs,maxl,ins,et)) ∧
        (∃C' mn pTs. (ins!pc = Invoke C' mn pTs ∨ ins!pc = Invoke_special C' mn pTs)
∧
        (mn,pTs) = sig0 ∧
        (∃apTs T ST'.
        (phi C sig)!pc = Some (((rev apTs) @ T # ST', LT),z) ∧
        length apTs = length pTs ∧
        (∃D' rT' body'. method (G,C') sig0 = Some(D',rT',body') ∧ G ⊢ rT0 ≼ rT') ∧
        (mn = init ⟶ constructor_ok G hp i (stk!length apTs) C' z0 r0) ∧
         correct_frame G hp i (ST, LT) maxl ins f ∧
         correct_frames G hp i phi rT sig z r frs))))"
```

Invariant for the whole program state:

constdefs

```
 correct_state :: "[jvm_prog,prog_type,jvm_state] ⇒ bool"
                 ("_,_ |-JVM _ [ok]"  [51,51] 50)
"correct_state G phi == λ(xp,hp,ihp,frs).
   case xp of
     None ⇒ (case frs of
                 [] ⇒ True
             | (f#fs) ⇒ G⊢h hp√ ∧ h_init G hp ihp ∧ preallocated hp ihp ∧
                      (let (stk,loc,C,sig,pc,r) = f
                       in ∃rT maxs maxl ins et s z.
                  is_class G C ∧
                  method (G,C) sig = Some(C,rT,(maxs,maxl,ins,et)) ∧
                  phi C sig ! pc = Some (s,z) ∧
                                correct_frame G hp ihp s maxl ins f ∧
                          correct_frames G hp ihp phi rT sig z r fs))
   | Some x ⇒ frs = []"
```

syntax *(xsymbols)*

```
 correct_state :: "[jvm_prog,prog_type,jvm_state] ⇒ bool"
                 ("_,_ ⊢JVM _ √"  [51,51] 50)
```

lemma *constructor_ok_field_update:*

```
  "⟦ constructor_ok G hp ihp x C' z r; hp a = Some(C,od) ⟧
  ⟹  constructor_ok G (hp(a↦(C, od(fl↦v)))) ihp x C' z r"
  apply (cases r)
  apply (unfold constructor_ok_def)
  apply (cases "∃y. x = Addr y")
   defer
   apply simp
  apply clarify
  apply simp
  apply (rule conjI)
   apply clarsimp
   apply blast
  apply (rule impI)
  apply (rule conjI)
   apply clarsimp
   apply blast
  apply (rule impI)
  apply (rule conjI)
   apply blast
  apply (rule impI, erule impE, assumption)
  apply (elim exE conjE)
  apply simp
  apply (rule exI)
  apply (rule conjI)
   defer
   apply (rule impI)
   apply (rule conjI)
    apply (rule refl)
   apply blast
  apply (rule impI)
  apply simp
  done

lemma constructor_ok_newref:
  "⟦ hp x = None; constructor_ok G hp ihp v C' z r ⟧
  ⟹ constructor_ok G (hp(x↦obj)) (ihp(x := T)) v C' z r"
apply (cases r)
apply (unfold constructor_ok_def)
apply clarify
apply (rule conjI)
 apply (rule refl)
apply clarsimp
apply (rule conjI)
 apply clarsimp
apply (rule impI)
apply (rule conjI)
 apply clarsimp
apply (rule impI)
apply (rule conjI)
 apply blast
apply (rule impI, erule impE, assumption)
apply (elim exE conjE)
apply (intro exI)
apply (rule conjI)
```

```
 defer
 apply (rule impI)
 apply (rule conjI, assumption)
 apply blast
apply clarsimp
done


lemma sup_ty_opt_OK:
  "(G ⊢ X <=o (OK T')) = (∃T. X = OK T ∧ G ⊢ T ≼i T')"
  apply (cases X)
  apply auto
  done


lemma constructor_ok_pass_val:
  "⟦ constructor_ok G hp ihp x C' True r;
     constructor_ok G hp ihp v C z0 r0; x = fst r0 ⟧
  ⟹ constructor_ok G hp ihp v C True (x, snd r)"
  apply (cases r, cases r0)
  apply (unfold constructor_ok_def)
  apply simp
  apply (elim conjE exE)
  apply simp
  apply (erule disjE)
   apply clarsimp
  apply clarsimp
  done


lemma sup_heap_newref:
  "hp oref = None ⟹ hp ≤| hp(oref ↦ obj)"
proof (unfold hext_def, intro strip)
  fix a C fs
  assume "hp oref = None" and hp: "hp a = Some (C, fs)"
  hence "a ≠ oref" by auto
  hence "(hp (oref↦obj)) a = hp a" by (rule fun_upd_other)
  with hp
  show "∃fs'. (hp(oref↦obj)) a = Some (C, fs')" by auto
qed

lemma sup_heap_update_value:
  "hp a = Some (C,od') ⟹ hp ≤| hp (a ↦ (C,od))"
by (simp add: hext_def)

lemma is_init_default_val:
  "is_init hp ihp (default_val T)"
  apply (simp add: is_init_def)
  apply (cases T)
   apply (case_tac prim_ty)
  apply auto
  done
```

4.22.1 approx-val

```
lemma iconf_widen:
  "G,hp,ihp ⊢ xcp ::≼i T ⟹ G ⊢ T ≼i T' ⟹ wf_prog wf_mb G
  ⟹ G,hp,ihp ⊢ xcp ::≼i T'"
  apply (cases T')
  apply (auto simp add: init_le_Init2 iconf_def elim: conf_widen)
  done

lemma is_init [elim!]:
  "G,hp,ihp ⊢ v ::≼i Init T ⟹ is_init hp ihp v"
  by (simp add: iconf_def)


lemma approx_val_Err:
  "approx_val G hp ihp x Err"
by (simp add: approx_val_def)

lemma approx_val_Null:
  "approx_val G hp ihp Null (OK (Init (RefT x)))"
  by (auto simp add: approx_val_def iconf_def is_init_def)


lemma approx_val_widen:
  "wf_prog wt G ⟹ approx_val G hp ihp v (OK T) ⟹ G ⊢ T ≼i T'
  ⟹ approx_val G hp ihp v (OK T')"
  by (unfold approx_val_def) (auto intro: iconf_widen)

lemma conf_notNone:
  "G,hp ⊢ Addr loc ::≼ ty ⟹ hp loc ≠ None"
  by (unfold conf_def) auto

lemma approx_val_imp_approx_val_sup_heap [rule_format]:
  "approx_val G hp ihp v at ⟶ hp ≤| hp' ⟶ approx_val G hp' ihp v at"
  apply (simp add: approx_val_def iconf_def is_init_def
              split: err.split init_ty.split)
  apply (fast dest: conf_notNone hext_objD intro: conf_hext)
  done



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_imp_approx_val_sup [rule_format]:
  "wf_prog wt G ⟹ (approx_val G h ih v us) ⟶ (G ⊢ us <=o us')
  ⟶ (approx_val G h ih v us')"
  apply (simp add: sup_PTS_eq approx_val_def iconf_def is_init_def
                   subtype_def init_le_Init
              split: err.split init_ty.split)
  apply (blast intro: conf_widen)
  done
```

lemma *approx_loc_imp_approx_val_sup:*
```
  "⟦ wf_prog wt G; approx_loc G hp ihp loc LT; idx < length LT;
      v = loc!idx; G ⊢ LT!idx <=o at ⟧
  ⟹ approx_val G hp ihp v at"
  apply (unfold approx_loc_def)
  apply (unfold list_all2_def)
  apply (auto intro: approx_val_imp_approx_val_sup
              simp add: split_def all_set_conv_all_nth)
  done
```

4.22.2 approx-loc

lemma *approx_loc_Cons [iff]:*
```
  "approx_loc G hp ihp (s#xs) (l#ls) =
  (approx_val G hp ihp s l ∧ approx_loc G hp ihp xs ls)"
  by (simp add: approx_loc_def)
```

lemma *approx_loc_Nil [simp,intro!]:*
```
  "approx_loc G hp ihp [] []"
  by (simp add: approx_loc_def)
```

lemma *approx_loc_len [elim]:*
```
  "approx_loc G hp ihp loc LT ⟹ length loc = length LT"
  by (unfold approx_loc_def list_all2_def) simp
```

lemma *approx_loc_replace_Err:*
```
  "approx_loc G hp ihp loc LT ⟹ approx_loc G hp ihp loc (replace v Err LT)"
  by (clarsimp simp add: approx_loc_def list_all2_conv_all_nth replace_def
                         approx_val_Err)
```

lemma *assConv_approx_stk_imp_approx_loc [rule_format]:*
```
  "wf_prog wt G ⟹ (∀(t,t')∈set (zip tys_n ts). G ⊢ t ≼i t')
  ⟶ length tys_n = length ts ⟶ approx_stk G hp ihp s tys_n ⟶
  approx_loc G hp ihp s (map OK ts)"
  apply (unfold approx_stk_def approx_loc_def list_all2_def)
  apply (clarsimp simp add: all_set_conv_all_nth)
  apply (rule approx_val_widen)
  apply auto
  done
```

lemma *approx_loc_imp_approx_loc_sup_heap:*
```
  "⟦ approx_loc G hp ihp lvars lt;  hp ≤| hp' ⟧
  ⟹ approx_loc G hp' ihp lvars lt"
  apply (unfold approx_loc_def list_all2_def)
  apply (auto intro: approx_val_imp_approx_val_sup_heap)
  done
```

lemma *approx_val_newref:*
```
"h loc = None ⟹
approx_val G (h(loc↦(C,fs))) (ih(loc:=UnInit C pc)) (Addr loc) (OK (UnInit C pc))"
  by (auto simp add: approx_val_def iconf_def is_init_def intro: conf_obj_AddrI)
```

```
lemma approx_val_newref_false:
  "⟦ h l = None; approx_val G h ih (Addr l) (OK T) ⟧ ⟹ False"
  by (simp add: approx_val_def iconf_def conf_def
           split: init_ty.split_asm ty.split_asm)

lemma approx_val_imp_approx_val_newref:
  "⟦ approx_val G hp ihp v T; hp loc = None ⟧
  ⟹ approx_val G hp (ihp(loc:=X)) v T"
  apply (cases "v = Addr loc")
  apply (auto simp add: approx_val_def iconf_def is_init_def conf_def
              split: err.split init_ty.split val.split)
  apply (erule allE, erule disjE)
  apply auto
  done

lemma approx_loc_newref:
 "⟦ approx_loc G hp ihp lvars lt;  hp loc = None ⟧
  ⟹ approx_loc G hp (ihp(loc:=X)) lvars lt"
  apply (unfold approx_loc_def list_all2_def)
  apply (auto intro: approx_val_imp_approx_val_newref)
  done

lemma approx_loc_newref_Err:
  "⟦ approx_loc G hp ihp loc LT; hp l = None; i < length loc; loc!i=Addr l ⟧
  ⟹ LT!i = Err"
  apply (cases "LT!i", simp)
  apply (clarsimp simp add: approx_loc_def list_all2_conv_all_nth)
  apply (erule allE, erule impE, assumption)
  apply (erule approx_val_newref_false)
  apply simp
  done


lemma approx_loc_newref_all_Err:
  "⟦ approx_loc G hp ihp loc LT; hp l = None ⟧
  ⟹ list_all2 (λv T. v = Addr l ⟶ T = Err) loc LT"
  apply (simp only: list_all2_conv_all_nth)
  apply (rule conjI)
   apply (simp add: approx_loc_def list_all2_def)
  apply clarify
  apply (rule approx_loc_newref_Err, assumption+)
  done

lemma approx_loc_imp_approx_loc_sup [rule_format]:
  "wf_prog wt G ⟹ approx_loc G hp ihp lvars lt ⟶ G ⊢ lt <=l lt'
  ⟶ approx_loc G hp ihp lvars lt'"
  apply (unfold Listn.le_def lesub_def sup_loc_def approx_loc_def list_all2_def)
  apply (auto simp add: all_set_conv_all_nth)
  apply (auto elim: approx_val_imp_approx_val_sup)
  done


lemma approx_loc_imp_approx_loc_subst [rule_format]:
  "∀ loc idx x X. (approx_loc G hp ihp loc LT) ⟶ (approx_val G hp ihp x X)
```

```
  ⟶ (approx_loc G hp ihp (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 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 ihp (replicate n v) (replicate n Err)"
  by (induct n) (auto simp add: approx_val_def)

lemmas [cong] = conj_cong

lemma approx_loc_append [rule_format]:
  "∀L1 l2 L2. length l1=length L1 ⟶
  approx_loc G hp ihp (l1@l2) (L1@L2) =
  (approx_loc G hp ihp l1 L1 ∧ approx_loc G hp ihp l2 L2)"
  apply (unfold approx_loc_def list_all2_def)
  apply simp
  apply blast
  done

lemmas [cong del] = conj_cong

lemma approx_val_Err_or_same:
  "⟦ approx_val G hp ihp v (OK X); X = UnInit C pc ∨ X = PartInit D;
     approx_val G hp ihp v X' ⟧
  ⟹ X' = Err ∨ X' = OK X"
  apply (simp add: approx_val_def split: err.split_asm)
  apply (erule disjE)
   apply (clarsimp simp add: iconf_def is_init_def split: init_ty.split_asm)
  apply (clarsimp simp add: iconf_def is_init_def split: init_ty.split_asm)
  done

lemma approx_loc_replace:
  "⟦ approx_loc G hp ihp loc LT; approx_val G hp ihp x' (OK X');
     approx_val G hp ihp x (OK X);
     X = UnInit C pc ∨ X = PartInit D; corr_loc loc LT ihp x X ⟧
  ⟹ approx_loc G hp ihp (replace x x' loc) (replace (OK X) (OK X') LT)"
  apply (simp add: approx_loc_def corr_loc_def replace_def list_all2_conv_all_nth)
  apply clarsimp
  apply (erule allE, erule impE, assumption)+
  apply simp
  apply (drule approx_val_Err_or_same, assumption+)
  apply (simp add: approx_val_Err)
  done
```

4.22.3 approx-stk

```
lemma list_all2_approx:
  "⋀s. list_all2 (approx_val G hp ihp) s (map OK S) =
       list_all2 (iconf G hp ihp) s S"
```

```
  apply (induct S)
  apply (auto simp add: list_all2_Cons2 approx_val_def)
  done

lemma list_all2_iconf_widen:
  "wf_prog mb G ⟹
  list_all2 (iconf G hp ihp) a b ⟹
  list_all2 (λx y. G ⊢ x ≼i Init y) b c ⟹
  list_all2 (λv T. G,hp ⊢ v ::≼ T ∧ is_init hp ihp v) a c"
  apply (rule list_all2_trans)
  defer
  apply assumption
  apply assumption
  apply (drule iconf_widen, assumption+)
  apply (simp add: iconf_def)
  done

lemma approx_stk_rev_lem:
  "approx_stk G hp ihp (rev s) (rev t) = approx_stk G hp ihp s t"
  apply (unfold approx_stk_def approx_loc_def list_all2_def)
  apply (auto simp add: zip_rev sym [OF rev_map])
  done

lemma approx_stk_rev:
  "approx_stk G hp ihp (rev s) t = approx_stk G hp ihp s (rev t)"
  by (auto intro: subst [OF approx_stk_rev_lem])

lemma approx_stk_imp_approx_stk_sup_heap [rule_format]:
  "∀lvars. approx_stk G hp ihp lvars lt ⟶ hp ≤| hp'
  ⟶ approx_stk G hp' ihp lvars lt"
  by (auto intro: approx_loc_imp_approx_loc_sup_heap simp add: approx_stk_def)

lemma approx_stk_imp_approx_stk_sup [rule_format]:
  "wf_prog wt G ⟹ approx_stk G hp ihp lvars st ⟶
  (G ⊢ map OK st <=l (map OK st'))
  ⟶ approx_stk G hp ihp lvars st'"
  by (auto intro: approx_loc_imp_approx_loc_sup simp add: approx_stk_def)

lemma approx_stk_Nil [iff]:
  "approx_stk G hp ihp [] []"
  by (simp add: approx_stk_def approx_loc_def)

lemma approx_stk_Cons [iff]:
  "approx_stk G hp ihp (x # stk) (S#ST) =
  (approx_val G hp ihp x (OK S) ∧ approx_stk G hp ihp stk ST)"
  by (simp add: approx_stk_def approx_loc_def)

lemma approx_stk_Cons_lemma [iff]:
  "approx_stk G hp ihp stk (S#ST') =
  (∃s stk'. stk = s#stk' ∧ approx_val G hp ihp s (OK S) ∧
  approx_stk G hp ihp stk' ST')"
  by (simp add: list_all2_Cons2 approx_stk_def approx_loc_def)

lemma approx_stk_len [elim]:
```

```
  "approx_stk G hp ihp stk ST ⟹ length stk = length ST"
  by (unfold approx_stk_def) (simp add: approx_loc_len)

lemma approx_stk_append_lemma:
  "approx_stk G hp ihp stk (S@ST') ⟹
   (∃s stk'. stk = s@stk' ∧ length s = length S ∧ length stk' = length ST' ∧
             approx_stk G hp ihp s S ∧ approx_stk G hp ihp stk' ST')"
  by (simp add: list_all2_append2 approx_stk_def approx_loc_def)

lemma approx_stk_newref:
 "⟦ approx_stk G hp ihp lvars lt;  hp loc = None ⟧
  ⟹ approx_stk G hp (ihp(loc:=X)) lvars lt"
  by (auto simp add: approx_stk_def intro: approx_loc_newref)

lemma newref_notin_stk:
  "⟦ approx_stk G hp ihp stk ST; hp l = None ⟧
  ⟹ Addr l ∉ set stk"
proof
  assume "Addr l ∈ set stk"
  then obtain a b where
    stk: "stk = a @ (Addr l) # b"
    by (clarsimp simp add: in_set_conv_decomp)
  hence "stk!(length a) = Addr l" by (simp add: nth_append)
  with stk obtain i where
    "stk!i = Addr l" and i: "i < length stk" by clarsimp
  moreover
  assume "approx_stk G hp ihp stk ST"
  hence l: "approx_loc G hp ihp stk (map OK ST)" by (unfold approx_stk_def)
  moreover
  assume "hp l = None"
  moreover
  from l
  have "length stk = length ST" by (simp add: approx_loc_def list_all2_def)
  with i
  have "map OK ST ! i ≠ Err" by simp
  ultimately
  show False
    by (auto dest: approx_loc_newref_Err)
qed
```

4.22.4 corresponds

```
lemma corr_loc_empty [simp]:
  "corr_loc [] [] ihp v T"
  by (simp add: corr_loc_def)

lemma corr_loc_cons:
  "corr_loc (s#loc) (L#LT) ihp v T =
   ((L = OK T ⟶ s = v ∧ corr_val v T ihp) ∧ corr_loc loc LT ihp v T)"
  by (simp add: corr_loc_def)

lemma corr_loc_start:
  "⋀loc.
  ⟦ ∀x ∈ set LT. x = Err ∨ (∃t. x = OK (Init t));
```

```
    length loc = length LT; T = UnInit C pc ∨ T = PartInit C'⟧
  ⟹ corr_loc loc LT ihp v T" (is "PROP ?P LT")
proof (induct LT)
  show "PROP ?P []" by (simp add: corr_loc_def)
  fix L LS assume IH: "PROP ?P LS"
  show "PROP ?P (L#LS)"
  proof -
    fix loc::"val list"
    assume "length loc = length (L # LS)"
    then obtain l ls where
      loc: "loc = l#ls" and len:"length ls = length LS"
      by (auto simp add: length_Suc_conv)
    assume "∀x∈set (L#LS). x = Err ∨ (∃t. x = OK (Init t))"
    then obtain
      first: "L = Err ∨ (∃t. L = OK (Init t))" and
      rest:  "∀x∈set LS. x = Err ∨ (∃t. x = OK (Init t))"
      by auto
    assume T: "T = UnInit C pc ∨ T = PartInit C'"
    with rest len IH
    have "corr_loc ls LS ihp v T" by blast
    with T first
    have "corr_loc (l#ls) (L # LS) ihp v T"
      by (clarsimp simp add: corr_loc_def)
    with loc
    show "corr_loc loc (L # LS) ihp v T" by simp
  qed
qed

lemma consistent_init_start:
  "⟦ LT0 = OK (Init (Class C)) # map OK (map Init pTs) @ replicate mxl' Err;
     loc = (oX # rev opTs @ replicate mxl' arbitrary);
     length pTs = length opTs ⟧
  ⟹ consistent_init [] loc ([], LT0) ihp"
  (is "⟦ PROP ?LT0; PROP ?loc; PROP _ ⟧ ⟹ PROP _")
proof -
  assume LT0: "PROP ?LT0" and "PROP ?loc" "length pTs = length opTs"
  hence len: "length loc = length LT0" by simp
  from LT0 have no_UnInit: "∀x ∈ set LT0. x = Err ∨ (∃t. x = OK (Init t))"
    by (auto dest: in_set_replicateD)
  with len show ?thesis
    by (simp add: consistent_init_def corresponds_def corr_stk_def)
       (blast intro: corr_loc_start)
qed

lemma corresponds_stk_cons:
  "corresponds (s#stk) loc (S#ST,LT) ihp v T =
  ((S = T ⟶ s = v ∧ corr_val v T ihp) ∧ corresponds stk loc (ST,LT) ihp v T)"
  by (simp add: corresponds_def corr_stk_def corr_loc_def)

lemma corresponds_loc_nth:
  "⟦ corresponds stk loc (ST,LT) ihp v T; n < length LT; LT!n = OK X ⟧ ⟹
  X = T ⟶ (loc!n) = v ∧ corr_val v T ihp"
  by (simp add: corresponds_def corr_loc_def list_all2_conv_all_nth)
```

```
lemma consistent_init_loc_nth:
  "⟦ consistent_init stk loc (ST,LT) ihp; n < length LT; LT!n = OK X ⟧
  ⟹ consistent_init (loc!n#stk) loc (X#ST,LT) ihp"
  apply (simp add: consistent_init_def corresponds_stk_cons)
  apply (intro strip conjI)
   apply (elim allE exE conjE)
   apply (intro exI conjI corresponds_loc_nth)
   apply assumption+
  apply (elim allE exE conjE)
  apply (intro exI conjI corresponds_loc_nth)
  apply assumption+
  done

lemma consistent_init_corresponds_stk_cons:
  "⟦ consistent_init (s#stk) loc (S#ST,LT) ihp;
     S = UnInit C pc ∨ S = PartInit C' ⟧
  ⟹ corresponds (s#stk) loc (S#ST, LT) ihp s S"
  by (simp add: consistent_init_def corresponds_stk_cons) blast

lemma consistent_init_corresponds_loc:
  "⟦ consistent_init stk loc (ST,LT) ihp; LT!n = OK T;
     T = UnInit C pc ∨ T = PartInit C'; n < length LT ⟧
  ⟹ corresponds stk loc (ST,LT) ihp (loc!n) T"
proof -
  assume "consistent_init stk loc (ST,LT) ihp"
         "T = UnInit C pc ∨ T = PartInit C'"
  then obtain v where
    corr: "corresponds stk loc (ST,LT) ihp v T"
    by (simp add: consistent_init_def) blast
  moreover
  assume "LT!n = OK T" "n < length LT"
  ultimately
  have "v = (loc!n)" by (simp add: corresponds_loc_nth)
  with corr
  show ?thesis by simp
qed

lemma consistent_init_pop:
  "consistent_init (s#stk) loc (S#ST,LT) ihp
  ⟹ consistent_init stk loc (ST,LT) ihp"
  by (simp add: consistent_init_def corresponds_stk_cons) fast

lemma consistent_init_Init_stk:
  "consistent_init stk loc (ST,LT) ihp ⟹
  consistent_init (s#stk) loc ((Init T')#ST,LT) ihp"
  by (simp add: consistent_init_def corresponds_def corr_stk_def corr_loc_def)

lemma corr_loc_set:
  "⟦ corr_loc loc LT ihp v T; OK T ∈ set LT ⟧ ⟹ corr_val v T ihp"
  by (auto simp add: corr_loc_def in_set_conv_decomp list_all2_append2
                     list_all2_Cons2)

lemma corresponds_var_upd_UnInit:
  "⟦ corresponds stk loc (ST,LT) ihp v T; n < length LT;
```

```
      OK T ∈ set (map OK ST) ∪ set LT ⟧
  ⟹ corresponds stk (loc[n:= v]) (ST,LT[n:= OK T]) ihp v T"
proof -
  assume "corresponds stk loc (ST,LT) ihp v T"
  then obtain
    stk: "corr_stk stk ST ihp v T" and
    loc: "corr_loc loc LT ihp v T"
    by (simp add: corresponds_def)
  from loc
  obtain
    len: "length loc = length LT" and
    "∀i. i < length loc ⟶ LT ! i = OK T ⟶ loc ! i = v ∧ corr_val v T ihp"
    (is "?all loc LT")
    by (simp add: corr_loc_def list_all2_conv_all_nth)
  moreover
  assume "OK T ∈ set (map OK ST) ∪ set LT"
  hence ihp: "corr_val v T ihp"
  proof
    assume "OK T ∈ set LT"
    with loc show ?thesis by (rule corr_loc_set)
  next
    assume "OK T ∈ set (map OK ST)"
    with stk show ?thesis by (unfold corr_stk_def) (rule corr_loc_set)
  qed
  moreover
  assume "n < length LT"
  ultimately
  have "?all (loc[n:= v]) (LT[n:= OK T])"
    by (simp add: nth_list_update)
  with len
  have "corr_loc (loc[n:= v]) (LT[n:= OK T]) ihp v T"
    by (simp add: corr_loc_def list_all2_conv_all_nth)
  with stk
  show ?thesis
    by (simp add: corresponds_def)
qed

lemma corresponds_var_upd_UnInit2:
  "⟦ corresponds stk loc (ST,LT) ihp v T; n < length LT; T' ≠ T ⟧
  ⟹ corresponds stk (loc[n:= v']) (ST,LT[n:= OK T']) ihp v T"
  by (auto simp add: corresponds_def corr_loc_def
                     list_all2_conv_all_nth nth_list_update)

lemma corresponds_var_upd:
 "⟦ corresponds (s#stk) loc (S#ST, LT) ihp v T; idx < length LT ⟧
  ⟹ corresponds stk (loc[idx := s]) (ST, LT[idx := OK S]) ihp v T"
apply (cases "S = T")
 apply (drule corresponds_var_upd_UnInit, assumption)
  apply simp
 apply (simp only: corresponds_stk_cons)
apply (drule corresponds_var_upd_UnInit2, assumption+)
apply (simp only: corresponds_stk_cons)
apply blast
done
```

```
lemma consistent_init_conv:
  "consistent_init stk loc s ihp =
   (∀T. ((∃C pc. T = UnInit C pc) ∨ (∃C. T = PartInit C)) ⟶
        (∃v. corresponds stk loc s ihp v T))"
  by (unfold consistent_init_def) blast

lemma consistent_init_store:
  "⟦ consistent_init (s#stk) loc (S#ST,LT) ihp; n < length LT ⟧
  ⟹ consistent_init stk (loc[n:= s]) (ST,LT[n:= OK S]) ihp"
apply (simp only: consistent_init_conv)
apply (blast intro: corresponds_var_upd)
done

lemma corr_loc_newT:
  "⟦ OK T ∉ set LT; length loc = length LT ⟧ ⟹ corr_loc loc LT ihp v T"
  by (auto dest: nth_mem simp add: corr_loc_def list_all2_conv_all_nth)

lemma corr_loc_new_val:
  "⟦ corr_loc loc LT ihp v T; Addr l ∉ set loc ⟧
  ⟹ corr_loc loc LT (ihp(l:= T')) v T"
proof -
  assume new: "Addr l ∉ set loc"
  assume "corr_loc loc LT ihp v T"
  then obtain
    len: "length loc = length LT" and
    all: "∀i. i < length loc ⟶ LT ! i = OK T ⟶ loc ! i = v ∧ corr_val v T ihp"
    by (simp add: corr_loc_def list_all2_conv_all_nth)
  show ?thesis
  proof (unfold corr_loc_def, simp only: list_all2_conv_all_nth,
         intro strip conjI)
    from len show "length loc = length LT" .
    fix i assume i: "i < length loc" and "LT!i = OK T"
    with all
    have l: "loc!i = v ∧ corr_val v T ihp" by blast
    thus "loc!i = v" by blast
    from i l new
    have "v ≠ Addr l" by (auto dest: nth_mem)
    with l
    show "corr_val v T (ihp(l:= T'))"
      by (auto simp add: corr_val_def split: init_ty.split)
  qed
qed

lemma corr_loc_Err_val:
  "⟦ corr_loc loc LT ihp v T; list_all2 (λv T. v = Addr l ⟶ T = Err) loc LT ⟧
  ⟹ corr_loc loc LT (ihp(l:= T')) v T"
  apply (simp add: corr_loc_def)
  apply (simp only: list_all2_conv_all_nth)
  apply clarify
  apply (simp (no_asm))
  apply clarify
  apply (erule_tac x = i in allE, erule impE)
```

```
   apply simp
   apply (erule impE, assumption)
  apply simp
  apply (unfold corr_val_def)
  apply (elim conjE exE)
  apply (rule_tac x = la in exI)
  apply clarsimp
  done

lemma corr_loc_len:
  "corr_loc loc LT ihp v T ⟹ length loc = length LT"
  by (simp add: corr_loc_def list_all2_conv_all_nth)

lemma corresponds_newT:
  "⟦ corresponds stk loc (ST,LT) ihp' v' T'; OK T ∉ set (map OK ST) ∪ set LT ⟧
  ⟹ corresponds stk loc (ST,LT) ihp v T"
  apply (clarsimp simp add: corresponds_def corr_stk_def)
  apply (rule conjI)
   apply (rule corr_loc_newT, simp)
   apply (simp add: corr_loc_len)
  apply (rule corr_loc_newT, assumption)
  apply (simp add: corr_loc_len)
  done

lemma corresponds_new_val:
  "⟦ corresponds stk loc (ST,LT) ihp v T; Addr l ∉ set stk;
     list_all2 (λv T. v = Addr l ⟶ T = Err) loc LT ⟧
  ⟹ corresponds stk loc (ST,LT) (ihp(l:= T')) v T"
  apply (simp add: corresponds_def corr_stk_def)
  apply (blast intro: corr_loc_new_val corr_loc_Err_val)
  done

lemma corresponds_newref:
  "⟦ corresponds stk loc (ST, LT) ihp v T;
     OK (UnInit C pc) ∉ set (map OK ST) ∪ set LT;
     Addr l ∉ set stk; list_all2 (λv T. v = Addr l ⟶ T = Err) loc LT ⟧
  ⟹ ∃v. corresponds (Addr l#stk) loc ((UnInit C pc)#ST,LT)
                    (ihp(l:= UnInit C pc)) v T"
  apply (simp add: corresponds_stk_cons)
  apply (cases "T = UnInit C pc")
   apply simp
   apply (rule conjI)
    apply (unfold corr_val_def)
    apply simp
   apply (rule corresponds_newT, assumption)
   apply simp
  apply (blast intro: corresponds_new_val)
  done

lemma consistent_init_new_val_lemma:
  "⟦ consistent_init stk loc (ST,LT) ihp;
     Addr l ∉ set stk; list_all2 (λv T. v = Addr l ⟶ T = Err) loc LT ⟧
  ⟹ consistent_init stk loc (ST,LT) (ihp(l:= T'))"
  by (unfold consistent_init_def) (blast intro: corresponds_new_val)
```

```
lemma consistent_init_newref_lemma:
  "⟦ consistent_init stk loc (ST,LT) ihp;
     OK (UnInit C pc) ∉ set (map OK ST) ∪ set LT;
     Addr l ∉ set stk; list_all2 (λv T. v = Addr l ⟶ T = Err) loc LT ⟧
  ⟹ consistent_init (Addr l#stk) loc ((UnInit C pc)#ST,LT) (ihp(l:= UnInit C pc))"
  by (unfold consistent_init_def) (blast intro: corresponds_newref)


lemma consistent_init_newref:
  "⟦ consistent_init stk loc (ST,LT) ihp;
     approx_loc G hp ihp loc LT;
     hp l = None;
     approx_stk G hp ihp stk ST;
     OK (UnInit C pc) ∉ set (map OK ST) ∪ set LT ⟧
  ⟹ consistent_init (Addr l#stk) loc ((UnInit C pc)#ST,LT) (ihp(l:= UnInit C pc))"
  apply (drule approx_loc_newref_all_Err, assumption)
  apply (drule newref_notin_stk, assumption)
  apply (rule consistent_init_newref_lemma)
  apply assumption+
  done

lemma corresponds_new_val2:
  "⟦corresponds stk loc (ST, LT) ihp v T; approx_loc G hp ihp loc LT; hp l = None;
   approx_stk G hp ihp stk ST⟧
  ⟹ corresponds stk loc (ST, LT) (ihp(l := T')) v T"
  apply (drule approx_loc_newref_all_Err, assumption)
  apply (drule newref_notin_stk, assumption)
  apply (rule corresponds_new_val)
  apply assumption+
  done

lemma consistent_init_new_val:
  "⟦ consistent_init stk loc (ST,LT) ihp;
     approx_loc G hp ihp loc LT; hp l = None;
     approx_stk G hp ihp stk ST ⟧
  ⟹ consistent_init stk loc (ST,LT) (ihp(l:= T'))"
  apply (drule approx_loc_newref_all_Err, assumption)
  apply (drule newref_notin_stk, assumption)
  apply (rule consistent_init_new_val_lemma)
  apply assumption+
  done

lemma UnInit_eq:
  "⟦ T = PartInit C' ∨ T = UnInit C pc; G ⊢ t <=o OK T ⟧ ⟹ t = OK T"
  apply (simp add: sup_ty_opt_def Err.le_def lesub_def split: err.splits)
  apply (case_tac a)
  apply (auto simp add: init_le_def)
  done

lemma corr_loc_widen:
  "⟦ corr_loc loc LT ihp v T; G ⊢ LT <=l LT'; T = PartInit C ∨ T = UnInit C pc ⟧
  ⟹ corr_loc loc LT' ihp v T"
  apply (simp add: list_all2_conv_all_nth sup_loc_length corr_loc_def)
```

```
  apply clarify
  apply (erule allE, erule impE)
   apply (simp add: sup_loc_length)
  apply (erule impE)
   apply (subgoal_tac "i < length LT")
    apply (drule sup_loc_nth)
    apply assumption
    apply (simp add: UnInit_eq)
   apply (simp add: sup_loc_length)
  apply assumption
  done

lemma corresponds_widen:
  "⟦ corresponds stk loc s ihp v T; G ⊢ s <=s s';
      T = PartInit C ∨ T = UnInit C pc ⟧
  ⟹ corresponds stk loc s' ihp v T"
  apply (cases s, cases s')
  apply (simp add: corresponds_def corr_stk_def sup_state_conv)
  apply (blast intro: corr_loc_widen)
  done

lemma corresponds_widen_split:
  "⟦ corresponds stk loc (ST,LT) ihp v T;
      G ⊢ map OK ST <=l map OK ST'; G ⊢ LT <=l LT';
      T = PartInit C ∨ T = UnInit C pc ⟧
  ⟹ corresponds stk loc (ST',LT') ihp v T"
  apply (drule_tac s = "(ST,LT)" and s' = "(ST',LT')" in corresponds_widen)
  apply (simp add: sup_state_conv )
  apply blast+
  done

lemma consistent_init_widen:
  "⟦ consistent_init stk loc s ihp; G ⊢ s <=s s' ⟧
  ⟹ consistent_init stk loc s' ihp"
  apply (simp add: consistent_init_def)
  apply (blast intro: corresponds_widen)
  done

lemma consistent_init_widen_split:
  "⟦ consistent_init stk loc (ST,LT) ihp;
      G ⊢ map OK ST <=l map OK ST'; G ⊢ LT <=l LT' ⟧
  ⟹ consistent_init stk loc (ST',LT') ihp"
proof -
  assume "consistent_init stk loc (ST,LT) ihp"
  moreover
  assume "G ⊢ map OK ST <=l map OK ST'" "G ⊢ LT <=l LT'"
  hence "G ⊢ (ST,LT) <=s (ST',LT')" by (simp add: sup_state_conv)
  ultimately
  show ?thesis by (rule consistent_init_widen)
qed

lemma corr_loc_replace_type:
  "⟦ corr_loc loc LT ihp v T; T ≠ (Init T'') ⟧
  ⟹ corr_loc loc (replace (OK T') (OK (Init T'')) LT) ihp v T"
```

```
  apply (unfold corr_loc_def replace_def)
  apply (simp add: list_all2_conv_all_nth)
  apply blast
  done

lemma corr_stk_replace_type:
  "⟦ corr_stk loc ST ihp v T; T ≠ (Init T'') ⟧
  ⟹ corr_stk loc (replace T' (Init T'') ST) ihp v T"
proof -
  assume "corr_stk loc ST ihp v T"
  hence "corr_loc loc (map OK ST) ihp v T" by (unfold corr_stk_def)
  moreover
  assume "T ≠ (Init T'')"
  ultimately
  have "corr_loc loc (replace (OK T') (OK (Init T'')) (map OK ST)) ihp v T"
    by (rule corr_loc_replace_type)
  moreover
  have "replace (OK T') (OK (Init T'')) (map OK ST) =
        map OK (replace T' (Init T'') ST)"
    by (rule replace_map_OK)
  ultimately
  have "corr_loc loc (map OK (replace T' (Init T'') ST)) ihp v T" by simp
  thus ?thesis by (unfold corr_stk_def)
qed

lemma consistent_init_replace_type:
  "⟦ consistent_init stk loc (ST,LT) ihp ⟧
  ⟹ consistent_init stk loc
      (replace T (Init T') ST, replace (OK T) (OK (Init T')) LT) ihp"
apply (unfold consistent_init_def corresponds_def)
apply simp
apply (blast intro: corr_stk_replace_type corr_loc_replace_type)
done

lemma corr_loc_replace:
  "⋀loc.
  ⟦ corr_loc loc LT ihp v T;  approx_loc G hp ihp loc LT;
     approx_val G hp ihp oX (OK T0); T ≠ T'; T = UnInit C pc ∨ T = PartInit D ⟧
  ⟹ corr_loc (replace oX x loc) (replace (OK T0) (OK T') LT) ihp v T"
  apply (frule corr_loc_len)
  apply (induct LT)
   apply (simp add: replace_def)
  apply (clarsimp simp add: length_Suc_conv)
  apply (clarsimp simp add: corr_loc_cons replace_Cons split del: split_if)
  apply (simp split: split_if_asm)
  apply clarify
  apply (drule approx_val_Err_or_same, assumption+)
  apply simp
  done

lemma corr_stk_replace:
  "⟦ corr_stk stk ST ihp v T; approx_stk G hp ihp stk ST;
      approx_val G hp ihp oX (OK T0); T ≠ T';
      T = UnInit C pc ∨ T = PartInit D ⟧
```

```
  ⟹ corr_stk (replace oX x stk) (replace T0 T' ST) ihp v T"
  apply (unfold corr_stk_def approx_stk_def)
  apply (drule corr_loc_replace, assumption+)
  apply (simp add: replace_map_OK)
  done

lemma corresponds_replace:
  "⟦ corresponds stk loc (ST,LT) ihp v T;
      approx_loc G hp ihp loc LT;
      approx_stk G hp ihp stk ST;
      approx_val G hp ihp oX (OK T0);
      T ≠ T'; T = UnInit C pc ∨ T = PartInit D ⟧
  ⟹ corresponds (replace oX x stk) (replace oX x loc)
                 (replace T0 T' ST, replace (OK T0) (OK T') LT) ihp v T"
  apply (clarsimp simp add: corresponds_def)
  apply (rule conjI)
   apply (rule corr_stk_replace)
   apply assumption+
  apply (rule corr_loc_replace)
  apply assumption+
  done

lemma consisten_init_replace:
  "⟦ consistent_init stk loc (ST,LT) ihp;
      approx_loc G hp ihp loc LT;
      approx_stk G hp ihp stk ST;
      approx_val G hp ihp oX (OK T0);
      T' = Init C' ⟧
  ⟹ consistent_init (replace oX x stk) (replace oX x loc)
                     (replace T0 T' ST, replace (OK T0) (OK T') LT) ihp"
  apply (simp only: consistent_init_conv)
  apply clarify apply (blast intro: corresponds_replace)
  done

lemma corr_loc_replace_Err:
  "corr_loc loc LT ihp v T ⟹ corr_loc loc (replace T' Err LT) ihp v T"
  by (simp add: corr_loc_def list_all2_conv_all_nth replace_def)

lemma corresponds_replace_Err:
  "corresponds stk loc (ST, LT) ihp v T
  ⟹ corresponds stk loc (ST, replace T' Err LT) ihp v T"
  by (simp add: corresponds_def corr_loc_replace_Err)

lemma consistent_init_replace_Err:
  "consistent_init stk loc (ST, LT) ihp
  ⟹ consistent_init stk loc (ST, replace T' Err LT) ihp"
  by (unfold consistent_init_def, blast intro: corresponds_replace_Err)

lemma corresponds_append:
  "⋀X. ⟦ corresponds (x@stk) loc (X@ST,LT) ihp v T; length X = length x ⟧
  ⟹ corresponds stk loc (ST,LT) ihp v T"
  apply (induct x)
   apply simp
  apply (clarsimp simp add: length_Suc_conv)
```

```
  apply (simp add: corresponds_stk_cons)
  apply blast
  done

lemma consistent_init_append:
  "⋀X. ⟦ consistent_init (x@stk) loc (X@ST,LT) ihp; length X = length x ⟧
  ⟹ consistent_init stk loc (ST,LT) ihp"
  apply (induct x)
   apply simp
  apply (clarsimp simp add: length_Suc_conv)
  apply (drule consistent_init_pop)
  apply blast
  done

lemma corresponds_xcp:
  "corresponds stk loc (ST, LT) ihp v T ⟹ T = UnInit C pc ∨ T = PartInit D
  ⟹ corresponds [x] loc ([Init X], LT) ihp v T"
  apply (simp add: corresponds_def corr_stk_def corr_loc_def)
  apply blast
  done

lemma consistent_init_xcp:
  "consistent_init stk loc (ST,LT) ihp
  ⟹ consistent_init [x] loc ([Init X], LT) ihp"
  apply (unfold consistent_init_def)
  apply (blast intro: corresponds_xcp)
  done

lemma corresponds_pop:
  "corresponds (s#stk) loc (S#ST,LT) ihp v T
  ⟹ corresponds stk loc (ST,LT) ihp v T"
  by (simp add: corresponds_stk_cons)

lemma consistent_init_Dup:
  "consistent_init (s#stk) loc (S#ST,LT) ihp
  ⟹ consistent_init (s#s#stk) loc (S#S#ST,LT) ihp"
  apply (simp only: consistent_init_conv)
  apply clarify
  apply (simp add: corresponds_stk_cons)
  apply blast
  done

lemma corresponds_Dup:
  "corresponds (s#stk) loc (S#ST,LT) ihp v T
  ⟹ corresponds (s#s#stk) loc (S#S#ST,LT) ihp v T"
  by (simp add: corresponds_stk_cons)

lemma corresponds_Dup_x1:
  "corresponds (s1#s2#stk) loc (S1#S2#ST,LT) ihp v T
  ⟹ corresponds (s1#s2#s1#stk) loc (S1#S2#S1#ST,LT) ihp v T"
  by (simp add: corresponds_stk_cons)

lemma consistent_init_Dup_x1:
  "consistent_init (s1#s2#stk) loc (S1#S2#ST,LT) ihp
```

```
  ⟹ consistent_init (s1#s2#s1#stk) loc (S1#S2#S1#ST,LT) ihp"
  by (unfold consistent_init_def, blast intro: corresponds_Dup_x1)

lemma corresponds_Dup_x2:
  "corresponds (s1#s2#s3#stk) loc (S1#S2#S3#ST,LT) ihp v T
  ⟹ corresponds (s1#s2#s3#s1#stk) loc (S1#S2#S3#S1#ST,LT) ihp v T"
  by (simp add: corresponds_stk_cons)

lemma consistent_init_Dup_x2:
  "consistent_init (s1#s2#s3#stk) loc (S1#S2#S3#ST,LT) ihp
  ⟹ consistent_init (s1#s2#s3#s1#stk) loc (S1#S2#S3#S1#ST,LT) ihp"
  by (unfold consistent_init_def, blast intro: corresponds_Dup_x2)

lemma corresponds_Swap:
  "corresponds (s1#s2#stk) loc (S1#S2#ST,LT) ihp v T
  ⟹ corresponds (s2#s1#stk) loc (S2#S1#ST,LT) ihp v T"
  by (simp add: corresponds_stk_cons)

lemma consistent_init_Swap:
  "consistent_init (s1#s2#stk) loc (S1#S2#ST,LT) ihp
  ⟹ consistent_init (s2#s1#stk) loc (S2#S1#ST,LT) ihp"
  by (unfold consistent_init_def, blast intro: corresponds_Swap)
```

4.22.5 oconf

```
lemma oconf_blank:
  "⟦ is_class G D; wf_prog wf_mb G ⟧ ⟹ G,hp ⊢ blank G D √"
  by (auto simp add: oconf_def blank_def dest: fields_is_type)

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_imp_oconf_heap_newref [rule_format]:
  "⟦hp oref = None; G,hp ⊢ obj √; G,hp ⊢ obj' √⟧ ⟹ G,hp(oref↦obj') ⊢ obj √"
  apply (unfold oconf_def lconf_def)
  apply simp
  apply (fast intro: conf_hext sup_heap_newref)
  done

lemma oconf_imp_oconf_heap_update [rule_format]:
  "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 simp
  apply (force intro: approx_val_heap_update)
  done
```

4.22.6 hconf

```
lemma hconf_imp_hconf_newref [rule_format]:
  "hp oref = None ⟶ G⊢h hp√ ⟶ G,hp⊢obj√ ⟶ G⊢h hp(oref↦obj)√"
  apply (simp add: hconf_def)
```

```
  apply (fast intro: oconf_imp_oconf_heap_newref)
  done

lemma hconf_imp_hconf_field_update [rule_format]:
  "map_of (fields (G, oT)) (F, D) = Some T ∧ hp oloc = Some(oT,fs) ∧
  G,hp⊢v::≼T ∧ G⊢h hp√ ⟶ G⊢h hp(oloc ↦ (oT, fs((F,D)↦v)))√"
  apply (simp add: hconf_def)
  apply (force intro: oconf_imp_oconf_heap_update oconf_field_update
               simp add: obj_ty_def)
  done
```

4.22.7 h-init

```
lemma h_init_field_update:
  "⟦ h_init G hp ihp; hp x = Some (C, fs); is_init hp ihp v ⟧
  ⟹ h_init G (hp(x↦(C,fs(X↦v)))) ihp"
  apply (unfold h_init_def o_init_def l_init_def)
  apply clarsimp
  apply (rule conjI)
   apply clarify
   apply (rule conjI)
    apply (clarsimp simp add: is_init_def)
   apply clarsimp
   apply (elim allE, erule impE, assumption, elim allE,
          erule impE, rule exI, assumption)
   apply (clarsimp simp add: is_init_def)
  apply clarsimp
  apply (elim allE, erule impE, assumption, elim allE,
         erule impE, rule exI, assumption)
  apply (clarsimp simp add: is_init_def)
 done

lemma h_init_newref:
   "⟦ hp r = None; G ⊢h hp √; h_init G hp ihp ⟧
  ⟹ h_init G (hp(r↦blank G X)) (ihp(r := T'))"
  apply (unfold h_init_def o_init_def l_init_def blank_def)
  apply clarsimp
  apply (rule conjI)
   apply clarsimp
   apply (simp add: init_vars_def map_of_map)
   apply (rule is_init_default_val)
  apply clarsimp
  apply (elim allE, erule impE, assumption)
  apply (elim allE, erule impE)
   apply fastsimp
  apply clarsimp
  apply (simp add: is_init_def)
  apply (drule hconfD, assumption)
  apply (drule oconf_objD, assumption)
  apply clarsimp
  apply (drule new_locD, assumption)
  apply simp
  done
```

4.22.8 preallocated

```
lemma preallocated_field_update:
  "⟦ map_of (fields (G, oT)) X = Some T; hp a = Some(oT,fs);
     G⊢h hp√; preallocated hp ihp ⟧
  ⟹ preallocated (hp(a ↦ (oT, fs(X↦v)))) ihp"
  apply (unfold preallocated_def)
  apply (rule allI)
  apply (erule_tac x=x in allE)
  apply (simp add: is_init_def)
  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 ihp"
  shows preallocated_newref: "preallocated (hp(oref↦obj)) (ihp(oref:=T)) "
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 is_init_def)
qed
```

4.22.9 constructor-ok

```
lemma correct_frames_ctor_ok:
  "correct_frames G hp ihp phi rT sig z r frs
  ⟹ frs = [] ∨ (fst sig = init ⟶ (∃a C'. constructor_ok G hp ihp a C' z r))"
  apply (cases frs)
   apply simp
  apply simp
  apply clarify
  apply simp
  apply blast
  done
```

4.22.10 correct-frame

```
lemma correct_frameE [elim?]:
  "correct_frame G hp ihp (ST,LT) maxl ins (stk, loc, C, sig, pc, r) ⟹
  (⟦approx_stk G hp ihp stk ST; approx_loc G hp ihp loc LT;
    consistent_init stk loc (ST, LT) ihp;
    fst sig = init ⟶
      corresponds stk loc (ST, LT) ihp (fst r) (PartInit C) ∧
      (∃l. fst r = Addr l ∧ hp l ≠ None ∧ (ihp l = PartInit C ∨ (∃C'. ihp l = Init
(Class C'))));
    pc < length ins; length loc = length (snd sig) + maxl + 1⟧
    ⟹ P)
```

```
  ⟹ P"
  apply (unfold correct_frame_def)
  apply fast
  done
```

4.22.11 correct-frames

```
lemmas [simp del] = fun_upd_apply
lemmas [split del] = split_if

lemma correct_frames_imp_correct_frames_field_update [rule_format]:
  "∀rT C sig z r. correct_frames G hp ihp phi rT sig z r frs ⟶
  hp a = Some (C,od) ⟶ map_of (fields (G, C)) fl = Some fd ⟶
  G,hp⊢v::≼fd  ⟶
  correct_frames G (hp(a ↦ (C, od(fl↦v)))) ihp phi rT sig z r frs"
  apply (induct frs)
   apply simp
  apply clarify
  apply simp
  apply clarify
  apply (unfold correct_frame_def)
  apply (simp (no_asm_use))
  apply clarify
  apply (intro exI conjI)
        apply assumption
       apply assumption+
  apply (rule impI)
  apply (rule constructor_ok_field_update)
   apply (erule impE, assumption)
   apply simp
  apply assumption
 apply (rule approx_stk_imp_approx_stk_sup_heap, assumption)
 apply (rule sup_heap_update_value, assumption)
apply (rule approx_loc_imp_approx_loc_sup_heap, assumption)
apply (rule sup_heap_update_value, assumption)
apply assumption+
apply (rule impI)
apply (erule impE, assumption, erule conjE)
apply (rule conjI)
 apply assumption
apply (elim exE conjE)
apply (rule exI)
apply (rule conjI)
 apply assumption
apply (rule conjI)
 apply (simp add: fun_upd_apply)
 apply (case_tac "l = a")
  apply simp
 apply simp
apply assumption+
apply blast
done

lemma correct_frames_imp_correct_frames_newref [rule_format]:
```

```
  "∀rT sig z r. hp x = None ⟶ correct_frames G hp ihp phi rT sig z r frs
  ⟶ correct_frames G (hp(x↦obj)) (ihp(x:=T)) phi rT sig z r frs"
  apply (induct frs)
   apply simp
  apply clarify
  apply simp
  apply clarify
  apply (unfold correct_frame_def)
  apply (simp (no_asm_use))
  apply clarify
  apply (intro exI conjI)
           apply assumption+
     apply (rule impI)
     apply (erule impE, assumption)
     apply (rule constructor_ok_newref, assumption)
     apply simp
    apply (drule approx_stk_newref, assumption)
    apply (rule approx_stk_imp_approx_stk_sup_heap, assumption)
    apply (rule sup_heap_newref, assumption)
   apply (drule approx_loc_newref, assumption)
   apply (rule approx_loc_imp_approx_loc_sup_heap, assumption)
   apply (rule sup_heap_newref, assumption)
  apply (rule consistent_init_new_val, assumption+)
 apply (rule impI, erule impE, assumption, erule conjE)
 apply (rule conjI)
  apply (rule corresponds_new_val2, assumption+)
 apply (elim conjE exE)
 apply (simp add: fun_upd_apply)
 apply (case_tac "l = x")
  apply simp
 apply simp
 apply assumption+
 apply blast
 done

lemmas [simp add] = fun_upd_apply
lemmas [split add] = split_if
```

4.22.12 correct-state

```
lemma correct_stateE [elim?]:
  "G,phi ⊢JVM (None, hp, ihp, (stk,loc,C,sig,pc,r)#frs)√ ⟹
   (⋀rT maxs maxl ins et ST LT z.
      ⟦G ⊢h hp √; h_init G hp ihp; preallocated hp ihp; is_class G C; method (G, C)
sig = Some (C, rT, maxs, maxl, ins, et);
          phi C sig ! pc = Some ((ST,LT), z); correct_frame G hp ihp (ST,LT) maxl ins
(stk, loc, C, sig, pc, r);
          correct_frames G hp ihp phi rT sig z r frs⟧ ⟹ P)
  ⟹ P"
  apply (unfold correct_state_def)
  apply simp
  apply blast
  done
```

```
lemma correct_stateE2 [elim?]:
  "G,phi ⊢JVM (None, hp, ihp, (stk,loc,C,sig,pc,r)#frs)√ ⟹
  method (G,C) sig = Some (C,rT,maxs,maxl,ins,et) ⟹
  (⋀ST LT z.
      ⟦G ⊢h hp √; h_init G hp ihp; preallocated hp ihp; is_class G C;
         phi C sig ! pc = Some ((ST,LT), z);
         correct_frame G hp ihp (ST,LT) maxl ins (stk, loc, C, sig, pc, r);
         correct_frames G hp ihp phi rT sig z r frs⟧ ⟹ P)
  ⟹ P"
  apply (erule correct_stateE)
  apply simp
  apply fast
  done

lemma correct_stateI [intro?]:
  "⟦G ⊢h hp √; h_init G hp ihp; preallocated hp ihp; is_class G C;
    method (G, C) sig = Some (C, rT, maxs, maxl, ins, et);
    phi C sig ! pc = Some (s', z);
    correct_frame G hp ihp s' maxl ins (stk, loc, C, sig, pc, r);
    correct_frames G hp ihp phi rT sig z r frs⟧
  ⟹ G,phi ⊢JVM (None, hp, ihp, (stk,loc,C,sig,pc,r)#frs)√"
  apply (unfold correct_state_def)
  apply (cases s')
  apply simp
  done

end
```

4.23 BV Type Safety Proof

```
theory BVSpecTypeSafe = Correct:
```

This theory contains proof that the specification of the bytecode verifier only admits type safe programs.

4.23.1 Preliminaries

Simp and intro setup for the type safety proof:

```
lemmas defs1 = correct_state_def correct_frame_def sup_state_conv wt_instr_def
              eff_def norm_eff_def eff_bool_def

lemmas correctE = correct_stateE2 correct_frameE

lemmas widen_rules[intro] = approx_val_widen
                            approx_loc_imp_approx_loc_sup
                            approx_stk_imp_approx_stk_sup

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, ihp, (stk,loc,C,sig,pc,r)#frs)√ ⟧
  ⟹ wt_instr (ins!pc) G C rT (phi C sig) maxs (fst sig = init) (length ins) et pc"
  apply (elim correct_stateE correct_frameE)
  apply simp
  apply (blast intro: wt_jvm_prog_impl_wt_instr)
  done
```

4.23.2 Exception Handling

Exceptions don't touch anything except the stack:

```
lemma exec_instr_xcpt:
  "(fst (exec_instr i G hp ihp stk vars Cl sig pc z frs) = Some xcp)
  = (∃stk'. exec_instr i G hp ihp stk vars Cl sig pc z frs =
           (Some xcp, hp, ihp, (stk', vars, Cl, sig, pc, z)#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
    [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 X pc et = Some handler
  ⟹ match G X pc et = [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, initialized 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; is_init hp ihp xcp;
      G,phi ⊢JVM (None, hp, ihp, f#frs)√ ⟧
  ⟹ G,phi ⊢JVM (find_handler G (Some xcp) hp ihp frs)√"
  (is "⋀f. ⟦ ?wt; ?adr; ?hp; ?init; ?correct (None, hp, ihp, 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 do not change in the induction:
  assume adr: ?adr
  assume hp: ?hp
  assume init: ?init

  — the assumption for the cons case:
  fix f f' frs'  assume cr: "?correct (None, hp, ihp, f#f'#frs')"

  — the induction hypothesis as produced by Isabelle, immediatly simplified with the fixed assump-
tions above
  assume "⋀f. ⟦ ?wt; ?adr; ?hp; ?init; ?correct (None, hp, ihp, f#frs') ⟧ ⟹ ?correct
(?find frs')"
  with wt adr hp init
  have IH: "⋀f. ?correct (None, hp, ihp, f#frs') ⟹ ?correct (?find frs')" by blast

  obtain stk loc C sig pc r where f' [simp]: "f' = (stk,loc,C,sig,pc,r)"
    by (cases f')

  from cr have cr': "?correct (None, hp, ihp, f'#frs')"
    by (auto simp add: correct_state_def)
```

```
  then obtain rT maxs maxl ins et where
    meth: "method (G,C) sig = Some (C,rT,maxs,maxl,ins,et)"
    by (fastsimp elim!: correct_stateE)

  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 C rT (phi C sig) maxs (fst sig = init) (length ins)
et pc"
      by (rule wt_jvm_prog_impl_wt_instr_cor)

    from cr meth
    obtain C' mn pts ST LT z where
      ins: "ins ! pc = Invoke C' mn pts ∨ ins ! pc = Invoke_special C' mn pts" and
      phi: "phi C sig ! pc = Some ((ST, LT),z)"
      by (simp add: correct_state_def) fast

    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 have "xcpt_names (ins ! pc, G, pc, et) = match_any G pc et" by auto
    with wti phi have
      "∀D∈set (match_any G pc et). the (?match D) < length ins ∧
      G ⊢ Some (([Init (Class D)], LT),z) <=' phi C sig!the (?match D)"
      by (simp add: wt_instr_def eff_def xcpt_eff_def) (fast dest!: bspec)
    with in_any match' obtain
      pc: "handler_pc < length ins"
      "G ⊢ Some (([Init (Class D)], LT),z) <=' phi C sig ! handler_pc"
      by auto
    then obtain ST' LT' where
      phi': "phi C sig ! handler_pc = Some ((ST',LT'),z)" and
      less: "G ⊢ ([Init (Class D)], LT) <=s (ST',LT')"
      by auto
```

```
    from cr' phi meth f'
    have "correct_frame G hp ihp (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 ihp loc LT" and
     csi: "consistent_init stk loc (ST, LT) ihp" and
     cor: "(fst sig = init ⟶
            corresponds stk loc (ST, LT) ihp (fst r) (PartInit C) ∧
           (∃l. fst r = Addr l ∧ hp l ≠ None ∧ (ihp l = PartInit C ∨ (∃C'. ihp l =
Init (Class C')))))"
      by (fastsimp elim!: correct_frameE)

    let ?f = "([xcp], loc, C, sig, handler_pc, r)"
    have "correct_frame G hp ihp (ST', LT') maxl ins ?f"
    proof -
      from wf less loc
      have "approx_loc G hp ihp 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 init
      have "G,hp,ihp ⊢ xcp ::≼i Init (Class D)" by (simp add: iconf_def)
      with wf less loc
      have "approx_stk G hp ihp [xcp] ST'"
        by (auto simp add: sup_state_conv approx_stk_def approx_val_def iconf_def subtype_def
            elim!: conf_widen split: Err.split init_ty.split)
      moreover
      note len pc phi' csi cor
      ultimately
      show ?thesis
        apply (simp add: correct_frame_def)
        apply (blast intro:  consistent_init_xcp consistent_init_widen
                             corresponds_widen corresponds_xcp)
        done
    qed

    with cr' match phi phi' meth
    show ?thesis by (unfold correct_state_def) auto
  qed
qed
```

The requirement of lemma *uncaught_xcpt_correct* (that the exception is a valid, initialized 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 ihp stk vars Cl sig pc r frs) = Some xcp;
     wt_instr (ins!pc) G Cl rT (phi C sig) maxs (fst sig=init) (length ins) et pc;
     G,phi ⊢JVM (None, hp, ihp, (stk,loc,C,sig,pc,r)#frs)√ ⟧
```

```
  ⟹ ∃adr T. xcp = Addr adr ∧ hp adr = Some T ∧ is_init hp ihp xcp"
  (is "⟦ ?xcpt; ?wt; ?correct ⟧ ⟹ _")
proof -
  note [simp] = split_beta raise_system_xcpt_def
  note [split] = split_if_asm option.split_asm

  assume wt: ?wt
  assume correct: ?correct hence pre: "preallocated hp ihp" ..

  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
    from correct obtain ST LT z where
      "phi C sig ! pc = Some ((ST, LT), z)"
      "approx_stk G hp ihp stk ST"
      by (blast elim: correct_stateE correct_frameE)
    with Throw wt xcpt pre show ?thesis
      apply (unfold wt_instr_def)
      apply (clarsimp simp add: approx_val_def iconf_def)
      apply (blast dest: non_npD)+
      done
  next
    case Invoke_special with xcpt pre
    show ?thesis by (auto dest: new_Addr_OutOfMemory dest!: preallocatedD)
  qed (auto dest!: preallocatedD)
qed


lemma cname_of_xcp [intro]:
  "⟦preallocated hp ihp; xcp = Addr (XcptRef x)⟧ ⟹ cname_of hp xcp = Xcpt x"
  by (auto elim: preallocatedE [of hp ihp x])

lemma prealloc_is_init [simp]:
  "preallocated hp ihp ⟹ is_init hp ihp (Addr (XcptRef x))"
  by (drule preallocatedD) blast
```

Finally we can state that the next state always conforms when an exception occured:

```
lemma xcpt_correct:
  "⟦ wt_jvm_prog G phi;
      method (G,C) sig = Some (C,rT,maxs,maxl,ins,et);
      wt_instr (ins!pc) G C rT (phi C sig) maxs (fst sig=init) (length ins) et pc;
      fst (exec_instr (ins!pc) G hp ihp stk loc C sig pc r frs) = Some xcp;
      Some state' = exec (G, None, hp, ihp, (stk,loc,C,sig,pc,r)#frs);
      G,phi ⊢JVM (None, hp, ihp, (stk,loc,C,sig,pc,r)#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 C rT (phi C sig) maxs (fst sig=init) (length ins) et
pc"
  assume xp: "fst (exec_instr (ins!pc) G hp ihp stk loc C sig pc r frs) = Some xcp"
  assume s': "Some state' = exec (G, None, hp, ihp, (stk,loc,C,sig,pc,r)#frs)"
  assume correct: "G,phi ⊢JVM (None, hp, ihp, (stk,loc,C,sig,pc,r)#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" "is_init hp ihp xcp"
    by (blast dest: exec_instr_xcpt_hp)
  moreover
  note correct
  ultimately
  have "G,phi ⊢JVM find_handler G (Some xcp) hp ihp 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 z where
      hp_ok:  "G ⊢h hp √" and
      h_ini:  "h_init G hp ihp" and
      prehp:  "preallocated hp ihp" and
      class:  "is_class G C" and
      phi_pc: "phi C sig ! pc = Some ((ST,LT),z)" and
      frame:  "correct_frame G hp ihp (ST, LT) maxl ins (stk, loc, C, sig, pc, r)" and
      frames: "correct_frames G hp ihp phi rT sig z r frs"
      ..

    from frame obtain
      stk: "approx_stk G hp ihp stk ST" and
      loc: "approx_loc G hp ihp loc LT" and
      pc:  "pc < length ins" and
      len: "length loc = length (snd sig)+maxl+1" and
      cin: "consistent_init stk loc (ST, LT) ihp" and
      cor: "fst sig = init ⟶
      corresponds stk loc (ST, LT) ihp (fst r) (PartInit C) ∧
      (∃l. fst r = Addr l ∧ hp l ≠ None ∧ (ihp l = PartInit C ∨ (∃C'. ihp l = Init
(Class C'))))"
      ..

    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, ihp, ([xcp], loc, C, sig, handler, r)#frs)"
    by (cases "ins!pc", auto simp add: raise_system_xcpt_def split_beta
                             split: split_if_asm) — takes long!

  let ?f' = "([xcp], loc, C, sig, handler, r)"
  from eff
  obtain ST' LT' where
    phi_pc': "phi C sig ! handler = Some ((ST', LT'),z)" and
    frame': "correct_frame G hp ihp (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'),z)" and
      less: "G ⊢ ([Init (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)
      with xcp prehp
      have "G,hp,ihp ⊢ xcp ::≼i Init (Class (Xcpt OutOfMemory))"
        by (simp add: iconf_def)
      moreover
      from wf less loc
      have "approx_loc G hp ihp loc LT'"
        by (simp add: sup_state_conv) blast
      moreover
      from wf less cin
      have "consistent_init [xcp] loc (ST',LT') ihp"
        by (blast intro: consistent_init_xcp consistent_init_widen)
      moreover
      note wf less pc' len cor
      ultimately
      have "correct_frame G hp ihp (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!: iconf_widen
                 intro: corresponds_xcp corresponds_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'), z)" and
      less: "G ⊢ ([Init (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,ihp ⊢ xcp ::≼i Init (Class (Xcpt NullPointer))"
        by (auto simp add: iconf_def conf_def obj_ty_def dest!: preallocatedD)
      moreover
      from wf less loc
      have "approx_loc G hp ihp loc LT'"
        by (simp add: sup_state_conv) blast
      moreover
      from wf less cin
      have "consistent_init [xcp] loc (ST', LT') ihp"
        by (blast intro: consistent_init_xcp consistent_init_widen)
      moreover
      note wf less pc' len cor
      ultimately
      have "correct_frame G hp ihp (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!: iconf_widen
                 intro: corresponds_xcp corresponds_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'), z)" and
      less: "G ⊢ ([Init (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,ihp ⊢ xcp ::≼i Init (Class (Xcpt NullPointer))"
```

```
        by (auto simp add: iconf_def conf_def obj_ty_def dest!: preallocatedD)
      moreover
      from wf less loc
      have "approx_loc G hp ihp loc LT'"
        by (simp add: sup_state_conv) blast
      moreover
      from wf less cin
      have "consistent_init [xcp] loc (ST', LT') ihp"
        by (blast intro: consistent_init_xcp consistent_init_widen)
      moreover
      note wf less pc' len cor
      ultimately
      have "correct_frame G hp ihp (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!: iconf_widen
                 intro: corresponds_xcp corresponds_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'), z)" and
      less: "G ⊢ ([Init (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,ihp ⊢ xcp ::≼i Init (Class (Xcpt ClassCast))"
        by (auto simp add: iconf_def conf_def obj_ty_def dest!: preallocatedD)
      moreover
      from wf less loc
      have "approx_loc G hp ihp loc LT'"
        by (simp add: sup_state_conv) blast
      moreover
      from wf less cin
      have "consistent_init [xcp] loc (ST', LT') ihp"
        by (blast intro: consistent_init_xcp consistent_init_widen)
      moreover
      note wf less pc' len cor
      ultimately
      have "correct_frame G hp ihp (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!: iconf_widen
                 intro: corresponds_xcp corresponds_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 (([Init (Class D)], LT),z) <=' 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 (([Init (Class D)], LT), z) <=' phi C sig ! handler"
      by auto
    then
    obtain ST' LT' where
      phi': "phi C sig ! handler = Some ((ST', LT'), z)" and
      less: "G ⊢ ([Init (Class D)], LT) <=s (ST', LT')"
      by auto
    from xp wt correct
    obtain addr T where
      xcp: "xcp = Addr addr" "hp addr = Some T" "is_init hp ihp xcp"
      by (blast dest: exec_instr_xcpt_hp)
    note phi'
    moreover
    { from xcp D
      have "G,hp,ihp ⊢ xcp ::≼i Init (Class D)"
        by (simp add: iconf_def conf_def obj_ty_def)
      moreover
      from wf less loc
      have "approx_loc G hp ihp loc LT'"
        by (simp add: sup_state_conv) blast
      moreover
      from wf less cin
      have "consistent_init [xcp] loc (ST', LT') ihp"
        by (blast intro: consistent_init_xcp consistent_init_widen)
      moreover
      note wf less pc' len cor
      ultimately
      have "correct_frame G hp ihp (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!: iconf_widen
                 intro: corresponds_xcp corresponds_widen)
    }
    ultimately
```

```
      show ?thesis by (rule that)
    next
      case Invoke_special
      with phi_pc eff
      have
        "∀D∈set (match_any G pc et).
        the (?m D) < length ins ∧ G ⊢ Some (([Init (Class D)], LT),z) <=' 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 (([Init (Class D)], LT), z) <=' phi C sig ! handler"
        by auto
      then
      obtain ST' LT' where
        phi': "phi C sig ! handler = Some ((ST', LT'), z)" and
        less: "G ⊢ ([Init (Class D)], LT) <=s (ST', LT')"
        by auto
      from xp wt correct
      obtain addr T where
        xcp: "xcp = Addr addr" "hp addr = Some T" "is_init hp ihp xcp"
        by (blast dest: exec_instr_xcpt_hp)
      note phi'
      moreover
      { from xcp D
        have "G,hp,ihp ⊢ xcp ::≼i Init (Class D)"
          by (simp add: iconf_def conf_def obj_ty_def)
        moreover
        from wf less loc
        have "approx_loc G hp ihp loc LT'"
          by (simp add: sup_state_conv) blast
        moreover
        from wf less cin
        have "consistent_init [xcp] loc (ST', LT') ihp"
          by (blast intro: consistent_init_xcp consistent_init_widen)
        moreover
        note wf less pc' len cor
        ultimately
        have "correct_frame G hp ihp (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!: iconf_widen
                   intro: corresponds_xcp corresponds_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 (([Init (Class D)], LT), z) <=' 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 (([Init (Class D)], LT), z) <=' phi C sig ! handler"
      by auto
    then
    obtain ST' LT' where
      phi': "phi C sig ! handler = Some ((ST', LT'), z)" and
      less: "G ⊢ ([Init (Class D)], LT) <=s (ST', LT')"
      by auto
    from xp wt correct
    obtain addr T where
      xcp: "xcp = Addr addr" "hp addr = Some T" "is_init hp ihp xcp"
      by (blast dest: exec_instr_xcpt_hp)
    note phi'
    moreover
    { from xcp D
      have "G,hp,ihp ⊢ xcp ::≼i Init (Class D)"
        by (simp add: iconf_def conf_def obj_ty_def)
      moreover
      from wf less loc
      have "approx_loc G hp ihp loc LT'"
        by (simp add: sup_state_conv) blast
      moreover
      note wf less pc' len
      from wf less cin
      have "consistent_init [xcp] loc (ST', LT') ihp"
        by (blast intro: consistent_init_xcp consistent_init_widen)
      moreover
      note wf less pc' len cor
      ultimately
      have "correct_frame G hp ihp (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!: iconf_widen
                 intro: corresponds_xcp corresponds_widen)
    }
    ultimately
    show ?thesis by (rule that)
```

```
    qed (insert xp', auto) — the other instructions do not generate exceptions

    from state' meth hp_ok class frames phi_pc' frame' h_ini prehp
    have ?thesis by simp (rule correct_stateI)
  }
  ultimately
  show ?thesis by (cases "?match") blast+
qed
```

4.23.3 Single Instructions

In this section we look at each single (welltyped) instruction, and prove that the state after execution of the instruction still conforms. Since we have already handled exceptions above, we can now assume, that on exception occurs for this (single step) execution.

```
lemmas [iff] = not_Err_eq

lemma Load_correct:
"⟦ wf_prog wt G;
    method (G,C) sig = Some (C,rT,maxs,maxl,ins,et);
    ins!pc = Load idx;
    wt_instr (ins!pc) G C rT (phi C sig) maxs (fst sig = init) (length ins) et pc;
    Some state' = exec (G, None, hp, ihp, (stk,loc,C,sig,pc,r)#frs);
    G,phi ⊢JVM (None, hp, ihp, (stk,loc,C,sig,pc,r)#frs)√ ⟧
⟹ G,phi ⊢JVM state'√"
apply (elim correctE, assumption)
apply (clarsimp simp add: defs1 map_eq_Cons)
apply (rule conjI)
 apply (rule approx_loc_imp_approx_val_sup)
 apply simp+
apply (rule conjI)
apply (blast intro: approx_stk_imp_approx_stk_sup
                    approx_loc_imp_approx_loc_sup)
apply (rule conjI)
apply (blast intro: approx_stk_imp_approx_stk_sup
                    approx_loc_imp_approx_loc_sup)
apply (rule conjI)
 apply (drule consistent_init_loc_nth)
  apply assumption+
 apply (erule consistent_init_widen_split)
  apply simp
  apply blast
 apply assumption
apply (rule impI, erule impE, assumption, elim exE conjE)
apply (simp add: corresponds_stk_cons)
apply (frule corresponds_loc_nth)
  apply assumption
 apply assumption
apply (rule conjI)
 apply (rule impI)
```

```
 apply (simp add: init_le_PartInit2)
apply (blast intro: corresponds_widen_split)
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 C rT (phi C sig) maxs (fst sig = init) (length ins) et pc;
  Some state' = exec (G, None, hp, ihp, (stk,loc,C,sig,pc,r)#frs);
  G,phi ⊢JVM (None, hp, ihp, (stk,loc,C,sig,pc,r)#frs)√ ⟧
⟹ G,phi ⊢JVM state'√"
apply (elim correctE, assumption)
apply (clarsimp simp add: defs1 map_eq_Cons)
apply (blast intro: approx_stk_imp_approx_stk_sup
                    approx_loc_imp_approx_loc_subst
                    approx_loc_imp_approx_loc_sup
                    consistent_init_store
                    consistent_init_widen_split
                    corresponds_widen_split
                    corresponds_var_upd)
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 C rT (phi C sig) maxs (fst sig = init) (length ins) et pc;
    Some state' = exec (G, None, hp, ihp, (stk,loc,C,sig,pc,r)#frs);
    G,phi ⊢JVM (None, hp, ihp, (stk,loc,C,sig,pc,r)#frs)√ ⟧
⟹ G,phi ⊢JVM state'√"
  apply (elim correctE, assumption)
  apply (clarsimp simp add: defs1 approx_val_def iconf_def init_le_Init
                            sup_PTS_eq map_eq_Cons)
  apply (rule conjI)
   apply (blast dest: conf_litval intro: conf_widen)
  apply (rule conjI)
   apply (clarsimp simp add: is_init_def split: val.split)
  apply (rule conjI)
   apply  (blast dest: approx_stk_imp_approx_stk_sup)
  apply (rule conjI)
   apply  (blast dest: approx_loc_imp_approx_loc_sup)
  apply (rule conjI)
   apply (drule consistent_init_Init_stk)
   apply (erule consistent_init_widen_split)
    apply (simp add: subtype_def)
```

```
    apply blast
   apply assumption
  apply (rule impI, erule impE, assumption, elim conjE exE)
  apply (simp add: corresponds_stk_cons)
  apply (blast intro: corresponds_widen_split)
  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 C rT (phi C sig) maxs (fst sig = init) (length ins) et pc;
    Some state' = exec (G, None, hp, ihp, (stk,loc,C,sig,pc,r)#frs);
    G,phi ⊢JVM (None, hp, ihp, (stk,loc,C,sig,pc,r)#frs)√;
    fst (exec_instr (ins!pc) G hp ihp stk loc C sig pc r frs) = None ⟧
⟹ G,phi ⊢JVM state'√"
apply (elim correctE, assumption)
apply (clarsimp simp add: defs2 map_eq_Cons approx_val_def wt_jvm_prog_def
                split: split_if_asm)
apply (rule conjI)
 apply (clarsimp simp add: iconf_def init_le_Init)
 apply (blast intro: Cast_conf2)
apply (rule conjI)
 apply (erule approx_stk_imp_approx_stk_sup)
 apply assumption+
apply (rule conjI)
 apply (erule approx_loc_imp_approx_loc_sup)
 apply assumption+
apply (rule conjI)
 apply (drule consistent_init_pop)
 apply (drule consistent_init_widen_split)
```

```
  apply assumption+
 apply (clarsimp simp add: init_le_Init)
 apply (erule consistent_init_Init_stk)
apply (clarsimp simp add: init_le_Init corresponds_stk_cons)
apply (blast intro: corresponds_widen_split)
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 C rT (phi C sig) maxs (fst sig = init) (length ins) et pc;
  Some state' = exec (G, None, hp, ihp, (stk,loc,C,sig,pc,r)#frs);
  G,phi ⊢JVM (None, hp, ihp, (stk,loc,C,sig,pc,r)#frs)√;
  fst (exec_instr (ins!pc) G hp ihp stk loc C sig pc r frs) = None ⟧
⟹ G,phi ⊢JVM state'√"
apply (elim correctE, assumption)
apply (clarsimp simp add: defs2 map_eq_Cons wt_jvm_prog_def approx_val_def
                          iconf_def init_le_Init2
                split: option.split split_if_asm)
apply (frule conf_widen)
apply assumption+
apply (drule conf_RefTD)
apply (clarsimp simp add: defs2 approx_val_def)
apply (rule conjI)
 apply (clarsimp simp add: iconf_def init_le_Init)
 apply (rule conjI)
  apply (drule widen_cfs_fields, assumption+)
  apply (simp add: hconf_def oconf_def lconf_def)
  apply (erule allE, erule allE, erule impE, assumption)
  apply (simp (no_asm_use))
  apply (erule allE, erule allE, erule impE, assumption)
  apply (elim exE conjE)
  apply simp
  apply (erule conf_widen, assumption+)
 apply (clarsimp simp: is_init_def split: val.split)
 apply (simp add: h_init_def o_init_def l_init_def)
 apply (erule allE, erule allE, erule impE, assumption)
 apply (drule hconfD, assumption)
 apply (drule widen_cfs_fields, assumption+)
 apply (drule oconf_objD, assumption)
 apply clarsimp
 apply (erule allE, erule impE) back apply blast
 apply (clarsimp simp add: is_init_def)
apply (clarsimp simp add: init_le_Init corresponds_stk_cons)
apply (blast intro: approx_stk_imp_approx_stk_sup
                    approx_loc_imp_approx_loc_sup
```

```
                    consistent_init_Init_stk
                    consistent_init_pop
                    consistent_init_widen_split
                    corresponds_widen_split)
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 C rT (phi C sig) maxs (fst sig = init) (length ins) et pc;
  Some state' = exec (G, None, hp, ihp, (stk,loc,C,sig,pc,r)#frs) ;
  G,phi ⊢JVM (None, hp, ihp, (stk,loc,C,sig,pc,r)#frs)√;
  fst (exec_instr (ins!pc) G hp ihp stk loc C sig pc r 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 = Putfield F D"
  assume wt:   "wt_instr (ins!pc) G C rT (phi C sig) maxs
                         (fst sig = init) (length ins) et pc"
  assume exec: "Some state' = exec (G, None, hp, ihp, (stk,loc,C,sig,pc,r)#frs)"
  assume conf: "G,phi ⊢JVM (None, hp, ihp, (stk,loc,C,sig,pc,r)#frs)√"
  assume no_x: "fst (exec_instr (ins!pc) G hp ihp stk loc C sig pc r frs) = None"

  from ins conf meth
  obtain ST LT z where
    heap_ok:    "G⊢h hp√" and
    pre_alloc:  "preallocated hp ihp" and
    init_ok:    "h_init G hp ihp" and
    phi_pc:     "phi C sig!pc = Some ((ST,LT),z)" and
    is_class_C: "is_class G C" and
    frame:      "correct_frame G hp ihp (ST,LT) maxl ins (stk,loc,C,sig,pc,r)" and
    frames:     "correct_frames G hp ihp phi rT sig z r frs"
    by (fastsimp elim: correct_stateE)

  from phi_pc ins wt
  obtain vT oT vT' tST ST' LT' where
    ST:      "ST = (Init vT) # (Init oT) # tST" and
    suc_pc:  "Suc pc < length ins" and
    phi_suc: "phi C sig ! Suc pc = Some ((ST',LT'),z)" and
    less:    "G ⊢ (tST, LT) <=s (ST', LT')" and
    class_D: "is_class G D"  and
    field:   "field (G, D) F = Some (D, vT')" and
    v_less:  "G ⊢ Init vT ⪯i Init vT'" and
    o_less:  "G ⊢ Init oT ⪯i Init (Class D)"
    by (unfold wt_instr_def eff_def eff_bool_def norm_eff_def)
```

```
    (clarsimp simp add: init_le_Init2, blast)

  from ST frame
  obtain vt ot stk' where
    stk  : "stk = vt#ot#stk'" and
    app_v: "approx_val G hp ihp vt (OK (Init vT))" and
    app_o: "approx_val G hp ihp ot (OK (Init oT))" and
    app_s: "approx_stk G hp ihp stk' tST" and
    app_l: "approx_loc G hp ihp loc LT" and
    consi: "consistent_init (vt#ot#stk') loc ((Init vT)#(Init oT)#tST,LT) ihp" and
    corr:  "fst sig = init ⟶
    corresponds (vt#ot#stk') loc (Init vT#Init oT#tST,LT) ihp (fst r) (PartInit C) ∧
    (∃l. fst r = Addr l ∧ hp l ≠ None ∧
    (ihp l = PartInit C ∨ (∃C'. ihp l = Init (Class C'))))" and
    len_l: "length loc = Suc (length (snd sig) + maxl)"
    by - (erule correct_frameE, simp, blast)

  from app_v app_o obtain
    "G,hp ⊢ vt ::≼ vT" and
    conf_o: "G,hp ⊢ ot ::≼ oT" and
    is_init_vo: "is_init hp ihp vt" "is_init hp ihp ot"
    by (simp add: approx_val_def iconf_def)

  with wf v_less have conf_v: "G,hp ⊢ vt ::≼ vT'"
    by (auto simp add: subtype_def intro: conf_widen)

  { assume "ot = Null"
    with exec ins meth stk obtain x where
      "fst (exec_instr (ins!pc) G hp ihp stk loc C sig pc r frs) = Some x"
      by (simp add: split_beta raise_system_xcpt_def)
    with no_x have ?thesis by simp
  }
  moreover
  { assume "ot ≠ Null"
    with o_less conf_o
    obtain x C' fs oT' where
      ot_addr: "ot = Addr x" and
      hp_Some: "hp x = Some (C',fs)" and
               "G ⊢ (Class C') ≼ oT"
      by (fastsimp simp add: subtype_def
                   dest: widen_RefT2 conf_RefTD)
    with o_less
    have C': "G ⊢ C' ≼C D"
      by (auto simp add: subtype_def dest: widen_trans)
    with wf field
    have fields: "map_of (fields (G, C')) (F, D) = Some vT'"
      by - (rule widen_cfs_fields)
```

```
    let ?hp' = "hp(x↦(C', fs((F, D)↦vt)))"
    and ?f'  = "(stk', loc, C, sig, Suc pc, r)"

    from exec ins meth stk ot_addr hp_Some
    have state': "state' = Norm (?hp', ihp, ?f'#frs)"
      by (simp add: raise_system_xcpt_def)

    from hp_Some have hext: "hp ≤| ?hp'" by (rule sup_heap_update_value)

    with fields hp_Some conf_v heap_ok
    have hp'_ok: "G ⊢h ?hp' √" by (blast intro: hconf_imp_hconf_field_update )

    from app_v have "is_init hp ihp vt" by (simp add: approx_val_def iconf_def)

    with init_ok hp_Some have hp'_init: "h_init G ?hp' ihp"
      by (rule h_init_field_update)

    from fields hp_Some heap_ok pre_alloc have pre_alloc': "preallocated ?hp' ihp"
      by (rule preallocated_field_update)

    from corr less have corr':
      "fst sig = init ⟶
      corresponds stk' loc (tST, LT) ihp (fst r) (PartInit C) ∧
      (∃l. fst r = Addr l ∧ ?hp' l ≠ None ∧
       (ihp l = PartInit C ∨ (∃C'. ihp l = Init (Class C'))))"
      by (clarsimp simp add: corresponds_stk_cons simp del: fun_upd_apply) simp

    from consi stk less have "consistent_init stk' loc (ST', LT') ihp"
      by (blast intro: consistent_init_pop consistent_init_widen)

    with wf app_l app_s len_l suc_pc less hext corr'
    have f'_correct:
      "correct_frame G ?hp' ihp (ST', LT') maxl ins (stk',loc,C,sig,Suc pc,r)"
      by (simp add: correct_frame_def sup_state_conv)
         (blast intro: approx_stk_imp_approx_stk_sup_heap
                       approx_stk_imp_approx_stk_sup
                       approx_loc_imp_approx_loc_sup_heap
                       approx_loc_imp_approx_loc_sup
                       corresponds_widen_split)

    from wf frames hp_Some fields conf_v
    have "correct_frames G ?hp' ihp phi rT sig z r frs"
      by - (rule  correct_frames_imp_correct_frames_field_update)

    with state' ins meth is_class_C phi_suc hp'_ok hp'_init f'_correct pre_alloc'
    have ?thesis by simp (rule correct_stateI)
  }
  ultimately
```

```
  show ?thesis by blast
qed


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 C rT (phi C sig) maxs (fst sig = init) (length ins) et pc;
  Some state' = exec (G, None, hp, ihp, (stk,loc,C,sig,pc,r)#frs) ;
  G,phi ⊢JVM (None, hp, ihp, (stk,loc,C,sig,pc,r)#frs)√;
  fst (exec_instr (ins!pc) G hp ihp stk loc C sig pc r 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 C rT (phi C sig) maxs
                         (fst sig = init) (length ins) et pc"
  assume exec: "Some state' = exec (G, None, hp, ihp, (stk,loc,C,sig,pc,r)#frs)"
  assume conf: "G,phi ⊢JVM (None, hp, ihp, (stk,loc,C,sig,pc,r)#frs)√"
  assume no_x: "fst (exec_instr (ins!pc) G hp ihp stk loc C sig pc r frs) = None"

  from conf meth
  obtain ST LT z where
    heap_ok:    "G⊢h hp√" and
    init_ok:    "h_init G hp ihp" and
    phi_pc:     "phi C sig!pc = Some ((ST,LT),z)" and
    prealloc:   "preallocated hp ihp" and
    is_class_C: "is_class G C" and
    frame:      "correct_frame G hp ihp (ST,LT) maxl ins (stk,loc,C,sig,pc,r)" and
    frames:     "correct_frames G hp ihp phi rT sig z r frs"
    ..

  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'),z)" and
    new_type:   "UnInit X pc ∉ set ST" and
    less: "G ⊢ (UnInit X pc#ST, replace (OK (UnInit X pc)) Err LT) <=s (ST', LT')"
      (is "G ⊢ (_,?LT) <=s (ST',LT')")
    by (unfold wt_instr_def eff_def eff_bool_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↦blank G X), ihp(oref := UnInit X pc),
                   (Addr oref # stk, loc, C, sig, Suc pc, r) # frs)"
    (is "state' = Norm (?hp', ?ihp', ?f # frs)")
    by simp
  moreover
  from wf hp heap_ok is_class_X
  have hp': "G ⊢h ?hp' √"
    by - (rule hconf_imp_hconf_newref,
          auto simp add: oconf_def blank_def dest: fields_is_type)
  moreover
  from hp heap_ok init_ok
  have "h_init G ?hp' ?ihp'" by (rule h_init_newref)
  moreover
  from hp have sup: "hp ≤| ?hp'" by (rule sup_heap_newref)
  from frame obtain
    cons:  "consistent_init stk loc (ST, LT) ihp" and
    a_loc: "approx_loc G hp ihp loc LT" and
    a_stk: "approx_stk G hp ihp stk ST" and
    corr:  "fst sig = init ⟶
      corresponds stk loc (ST, LT) ihp (fst r) (PartInit C) ∧
      (∃l. fst r = Addr l ∧ hp l ≠ None ∧
       (ihp l = PartInit C ∨ (∃C'. ihp l = Init (Class C'))))"
    ..
  from a_loc
  have a_loc': "approx_loc G hp ihp loc ?LT" by (rule approx_loc_replace_Err)
  from corr a_loc a_stk new_type hp
  have corr':
    "fst sig = init ⟶
     corresponds (Addr oref#stk) loc (UnInit X pc#ST, ?LT) ?ihp' (fst r) (PartInit C)
∧
     (∃l. fst r = Addr l ∧ ?hp' l ≠ None ∧
      (?ihp' l = PartInit C ∨ (∃C'. ?ihp' l = Init (Class C'))))"
    apply (clarsimp simp add: corresponds_stk_cons simp del: fun_upd_apply)
    apply (drule corresponds_new_val2, assumption+)
    apply (auto intro: corresponds_replace_Err)
    done
  from new_type have "OK (UnInit X pc) ∉ set (map OK ST) ∪ set ?LT"
    by (auto simp add: replace_removes_elem)
  with cons a_stk a_loc' hp
  have "consistent_init (Addr oref # stk) loc (UnInit X pc#ST,?LT) ?ihp'"
    by (blast intro: consistent_init_newref consistent_init_replace_Err)
  from this less have "consistent_init (Addr oref # stk) loc (ST',LT') ?ihp'"
```

```
    by (rule consistent_init_widen)
  with hp frame less suc_pc wf corr' a_loc'
  have "correct_frame G ?hp' ?ihp' (ST', LT') maxl ins ?f"
    apply (unfold correct_frame_def sup_state_conv)
    apply (clarsimp simp add: map_eq_Cons conf_def blank_def
                              corresponds_stk_cons)
    apply (insert sup, unfold blank_def)
    apply (blast intro: corresponds_widen_split
                        approx_stk_imp_approx_stk_sup
                        approx_stk_newref
                        approx_stk_imp_approx_stk_sup_heap
                        approx_loc_imp_approx_loc_sup_heap
                        approx_loc_imp_approx_loc_sup
                        approx_loc_newref
                        approx_val_newref
                        sup)
    done
  moreover
  from hp frames wf heap_ok is_class_X
  have "correct_frames G ?hp' ?ihp' phi rT sig z r frs"
    by (unfold blank_def)
      (rule correct_frames_imp_correct_frames_newref,
        auto simp add: oconf_def dest: fields_is_type)
  moreover
  from hp prealloc have "preallocated ?hp' ?ihp'" by (rule preallocated_newref)
  ultimately
  show ?thesis by simp (rule correct_stateI)
qed
```

Method Invocation

```
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 C rT (phi C sig) maxs (fst sig = init) (length ins) et pc;
  Some state' = exec (G, None, hp, ihp, (stk,loc,C,sig,pc,r)#frs) ;
  G,phi ⊢JVM (None, hp, ihp, (stk,loc,C,sig,pc,r)#frs)√;
  fst (exec_instr (ins!pc) G hp ihp stk loc C sig pc r 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 C rT (phi C sig) maxs
                           (fst sig = init) (length ins) et pc"
  assume state': "Some state' = exec (G, None, hp, ihp, (stk,loc,C,sig,pc,r)#frs)"
```

```
assume approx: "G,phi ⊢JVM (None, hp, ihp, (stk,loc,C,sig,pc,r)#frs)√"
assume no_xcp: "fst (exec_instr (ins!pc) G hp ihp stk loc C sig pc r frs) = None"

from wtprog obtain wfmb where
  wfprog: "wf_prog wfmb G" by (simp add: wt_jvm_prog_def)

from approx method obtain s z where
  heap_ok: "G⊢h hp√" and
  init_ok: "h_init G hp ihp" and
  phi_pc:  "phi C sig!pc = Some (s,z)" and
  prealloc:"preallocated hp ihp" and
  is_class_C: "is_class G C" and
  frame:   "correct_frame G hp ihp s maxl ins (stk, loc, C, sig, pc, r)" and
  frames:  "correct_frames G hp ihp phi rT sig z r frs"
  ..

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
  w: "∀(x,y)∈set (zip apTs pTs). G ⊢ x ⪯i (Init y)" 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 pc (Some (s,z)) <=' phi C sig!Suc pc" and
  X: "G ⊢ X ⪯i Init (Class C')" and
  ni: "mn ≠ init"
  by (simp add: wt_instr_def eff_def) blast

from eff obtain ST' LT' z' where
  s': "phi C sig ! Suc pc = Some ((ST',LT'),z')"
  by (simp add: norm_eff_def split_paired_Ex) fast

from s ins frame obtain
  a_stk: "approx_stk G hp ihp stk (rev apTs @ X # ST)" and
  a_loc: "approx_loc G hp ihp loc LT" and
  init:  "consistent_init stk loc s ihp" 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 ihp opTs (rev apTs)" and
  oX:     "approx_val G hp ihp oX (OK X)" and
  a_stk': "approx_stk G hp ihp stk' ST" and
  stk':   "stk = opTs @ oX # stk'" and
  l_o:    "length opTs = length apTs"
          "length stk' = length ST"
  by (auto dest!:  approx_stk_append_lemma)
```

```
from oX have X_oX: "G,hp,ihp ⊢ oX ::≼i X" by (simp add: approx_val_def)
with wfprog X have oX_conf: "G,hp ⊢ oX ::≼ (Class C')"
  by (auto simp add: approx_val_def iconf_def init_le_Init2 dest: conf_widen)
from stk' l_o l have oX_pos: "stk ! length pTs = oX" by (simp add: nth_append)
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  obtain D fs where
  hp_Some: "hp ref = Some (D,fs)" and
  D_le_C': "G ⊢ D ≼C C'"
  by (fastsimp dest: conf_RefTD)

from D_le_C' wfprog mC'
obtain D'' rT' mxl' mxs' ins' et' where
  mD: "method (G, D) (mn, pTs) = Some (D'', rT', mxs', mxl', ins', et')"
      "G ⊢ rT' ≼ rT"
  by (auto dest!: subtype_widen_methd) blast

let ?loc' = "oX # rev opTs @ replicate mxl' arbitrary"
let ?f    = "([], ?loc', D'', (mn, pTs), 0, arbitrary)"
let ?f'   = "(stk, loc, C, sig, pc, r)"

from oX_Addr oX_pos hp_Some state' method ins stk' l_o l mD
have state'_val: "state' = Norm (hp, ihp, ?f# ?f' # frs)"
  by (simp add: raise_system_xcpt_def)

from is_class D_le_C' 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 wtprog mD''
have start: "wt_start G D'' mn pTs mxl' (phi D'' (mn, pTs)) ∧ ins' ≠ []"
  by (auto dest: wt_jvm_prog_impl_wt_start)

then obtain LT0 z0 where
  LT0: "phi D'' (mn, pTs) ! 0 = Some (([], LT0),z0)"
  by (clarsimp simp add: wt_start_def sup_state_conv)

have c_f: "correct_frame G hp ihp ([], LT0) mxl' ins' ?f"
proof -
  let ?T = "OK (Init (Class D''))"
  from ni start LT0 have sup_loc:
    "G ⊢ (?T # map OK (map Init 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 ihp (replicate mxl' arbitrary) (replicate mxl' Err)"
      by (simp add: approx_loc_def approx_val_Err 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 hp_Some oX_Addr oX X have a: "approx_val G hp ihp oX ?T"
      by (auto simp add: is_init_def init_le_Init2
                         approx_val_def iconf_def conf_def)

    from w l
    have "∀(x,y)∈set (zip (map (λx. x) apTs) (map Init pTs)). G ⊢ x ≼i y"
      by (simp only: zip_map) auto
    hence "∀(x,y)∈set (zip apTs (map Init pTs)). G ⊢ x ≼i y" by simp
    with l
    have "∀(x,y)∈set (zip (rev apTs) (rev (map Init pTs))). G ⊢ x ≼i y"
      by (auto simp add: zip_rev)
    with wfprog l l_o opTs
    have "approx_loc G hp ihp opTs (map OK (rev (map Init pTs)))"
      by (auto intro: assConv_approx_stk_imp_approx_loc)
    hence "approx_stk G hp ihp opTs (rev (map Init pTs))"
      by (simp add: approx_stk_def)
    hence "approx_stk G hp ihp (rev opTs) (map Init pTs)"
      by (simp add: approx_stk_rev)
    hence "approx_loc G hp ihp (rev opTs) (map OK (map Init pTs))"
      by (simp add: approx_stk_def)
    with r a l_o l
    have "approx_loc G hp ihp ?loc' ?LT"
      by (auto simp add: approx_loc_append approx_stk_def)

    from wfprog this sup_loc have loc: "approx_loc G hp ihp ?loc' LT0"
      by (rule approx_loc_imp_approx_loc_sup)

    from l l_o have "length pTs = length opTs" by auto
    hence "consistent_init [] ?loc' ([],?LT) ihp"
      by (blast intro: consistent_init_start)
    with sup_loc have "consistent_init [] ?loc' ([],LT0) ihp"
      by (auto intro: consistent_init_widen_split)

    with start loc l_o l ni  show ?thesis by (simp add: correct_frame_def)
  qed

  from X X_oX oX_Addr hp_Some obtain X' where X': "X = Init (Class X')"
    by (auto simp add: init_le_Init2 iconf_def conf_def dest!: widen_Class)

  with X mC' wf obtain mD'' rT'' b'' where
```

```
    "method (G, X') (mn, pTs) = Some (mD'', rT'', b'')"
    "G ⊢ rT'' ≼ rT"
    by (simp add: subtype_def) (drule subtype_widen_methd, assumption+, blast)

  with X' state'_val heap_ok mD'' ins method phi_pc s l
    frames s' LT0 c_f frame is_class_C ni init_ok prealloc
  show "G,phi ⊢JVM state'√"
    apply simp
    apply (rule correct_stateI, assumption+)
    apply clarsimp
    apply (intro exI conjI impI)
       apply (rule refl)
      apply blast
     apply assumption
    apply assumption
    done
qed

lemma Invoke_special_correct:
"⟦ wt_jvm_prog G phi;
  method (G,C) sig = Some (C,rT,maxs,maxl,ins,et);
  ins ! pc = Invoke_special C' mn pTs;
  wt_instr (ins!pc) G C rT (phi C sig) maxs (fst sig = init) (length ins) et pc;
  Some state' = exec (G, None, hp, ihp, (stk,loc,C,sig,pc,r)#frs) ;
  G,phi ⊢JVM (None, hp, ihp, (stk,loc,C,sig,pc,r)#frs)√;
  fst (exec_instr (ins!pc) G hp ihp stk loc C sig pc r 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_special C' mn pTs"
  assume wti:    "wt_instr (ins!pc) G C rT (phi C sig) maxs
                          (fst sig = init) (length ins) et pc"
  assume state': "Some state' = exec (G, None, hp, ihp, (stk,loc,C,sig,pc,r)#frs)"
  assume approx: "G,phi ⊢JVM (None, hp, ihp, (stk,loc,C,sig,pc,r)#frs)√"
  assume no_x:   "fst (exec_instr (ins!pc) G hp ihp stk loc C sig pc r frs) = None"

  from wtprog obtain wfmb where wfprog: "wf_prog wfmb G"
    by (simp add: wt_jvm_prog_def)

  from approx method obtain s z where
    heap_ok: "G⊢h hp√" and
    init_ok: "h_init G hp ihp" and
    prealloc:"preallocated hp ihp" and
    phi_pc:  "phi C sig!pc = Some (s,z)" and
    is_class_C: "is_class G C" and
    frame:   "correct_frame G hp ihp s maxl ins (stk, loc, C, sig, pc, r)" and
    frames:  "correct_frames G hp ihp phi rT sig z r frs"
```

```
  ..

from ins wti phi_pc obtain apTs X ST LT rT' maxs' mxl' ins' et' where
  s: "s = (rev apTs @ X # ST, LT)" and
  l: "length apTs = length pTs" and
  is_class: "is_class G C'" and
  w: "∀(x,y)∈set (zip apTs pTs). G ⊢ x ⪯i (Init y)" and
  mC': "method (G, C') (mn, pTs) = Some (C', rT', maxs', mxl', ins', et')" and
  pc:  "Suc pc < length ins" and
  eff: "G ⊢ norm_eff (Invoke_special C' mn pTs) G pc (Some (s,z))
           <=' 
           phi C sig!Suc pc" and
  X: "(∃pc. X = UnInit C' pc) ∨ (X = PartInit C ∧ G ⊢ C ≺C1 C' ∧ ¬z)" and
  is_init: "mn = init"
  by (simp add: wt_instr_def split_paired_Ex eff_def) blast

from s eff obtain ST' LT' z' where
  s': "phi C sig ! Suc pc = Some ((ST',LT'),z')"
  by (clarsimp simp add: norm_eff_def) blast

from s ins frame obtain
  a_stk: "approx_stk G hp ihp stk (rev apTs @ X # ST)" and
  a_loc: "approx_loc G hp ihp loc LT" and
  init:  "consistent_init stk loc (rev apTs @ X # ST, LT) ihp" and
  corr:  "fst sig = init ⟶ corresponds stk loc s ihp (fst r) (PartInit C) ∧
          (∃l. fst r = Addr l ∧ hp l ≠ None ∧
            (ihp l = PartInit C ∨ (∃C'. ihp l = Init (Class C'))))" 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 ihp opTs (rev apTs)" and
  oX:     "approx_val G hp ihp oX (OK X)" and
  a_stk': "approx_stk G hp ihp stk' ST" and
  stk':   "stk = opTs @ oX # stk'" and
  l_o:    "length opTs = length apTs"
          "length stk' = length ST"
  by (fastsimp dest!: approx_stk_append_lemma)

from stk' l_o l have oX_pos: "stk ! length pTs = oX" by (simp add: nth_append)

from state' method ins no_x oX_pos have "oX ≠ Null"
  by (simp add: raise_system_xcpt_def split: split_if_asm)
moreover
from wfprog X oX have oX_conf: "G,hp ⊢ oX ::⪯ (Class C')"
  by (auto simp add: approx_val_def iconf_def)
     (blast dest: conf_widen)
ultimately
```

```
obtain ref obj D fs where
  oX_Addr: "oX = Addr ref" and
  hp_Some: "hp ref = Some (D,fs)" and
  D_le_C': "G ⊢ D ≼C C'"
  by (fastsimp dest: conf_RefTD)

let ?new  = "new_Addr hp"
let ?ref' = "fst ?new"
let ?xp'  = "snd ?new"
let ?hp'  = "hp(?ref'↦blank G D)"
let ?loc' = "Addr ?ref' # rev opTs @ replicate mxl' arbitrary"
let ?ihp' = "if C' = Object then
             ihp(?ref':= Init (Class D))
             else
             ihp(?ref' := PartInit C')"
let ?r'   = "if C' = Object then
             (Addr ?ref', Addr ?ref')
             else
             (Addr ?ref', Null)"
let ?f    = "([], ?loc', C', (mn, pTs), 0, ?r')"
let ?f'   = "(stk, loc, C, sig, pc, r)"

from state' method ins no_x have norm: "?xp' = None"
  by (simp add: split_beta split: split_if_asm)

with oX_Addr oX_pos hp_Some state' method ins stk' l_o l mC'
have state'_val: "state' = Norm (?hp', ?ihp', ?f# ?f' # frs)"
  by (simp add: raise_system_xcpt_def split_beta)

obtain ref' xp' where
  new_Addr: "new_Addr hp = (ref',xp')"
  by (cases "new_Addr hp") auto
with norm have new_ref': "hp ref' = None" by (auto dest: new_AddrD)

from is_class D_le_C' have is_class_D: "is_class G D"
  by (auto dest: subcls_is_class2)

from wtprog is_class mC'
have start: "wt_start G C' mn pTs mxl' (phi C' (mn, pTs)) ∧ ins' ≠ []"
  by (auto dest: wt_jvm_prog_impl_wt_start)

then obtain LT0 where
  LT0: "phi C' (mn, pTs) ! 0 = Some (([], LT0),C'=Object)"
  by (clarsimp simp add: wt_start_def sup_state_conv)

let ?T = "OK (if C' ≠ Object then PartInit C' else Init (Class C'))"
from start LT0 is_init
have sup_loc:
```

```
    "G ⊢ (?T # map OK (map Init pTs) @ replicate mxl' Err) <=l LT0"
    (is "G ⊢ ?LT <=l LT0")
    by (simp add: wt_start_def sup_state_conv)

  have c_f: "correct_frame G ?hp' ?ihp' ([], LT0) mxl' ins' ?f"
  proof -
    have r:
      "approx_loc G ?hp' ?ihp' (replicate mxl' arbitrary) (replicate mxl' Err)"
      by (simp add: approx_loc_def approx_val_Err list_all2_def
                   set_replicate_conv_if)

    from new_Addr obtain fs where "?hp' ref' = Some (D,fs)" by (simp add: blank_def)
    hence "G,?hp' ⊢ Addr ref' ::≼ Class C'"
      by (auto simp add: new_Addr D_le_C' intro: conf_obj_AddrI)
    hence a: "approx_val G ?hp' ?ihp' (Addr ?ref') ?T"
      by (auto simp add: approx_val_def iconf_def new_Addr
                         blank_def is_init_def)
    from w l
    have "∀(x,y)∈set (zip (map (λx. x) apTs) (map Init pTs)). G ⊢ x ≼i y"
      by (simp only: zip_map) auto
    hence "∀(x,y)∈set (zip apTs (map Init pTs)). G ⊢ x ≼i y" by simp
    with l
    have "∀(x,y)∈set (zip (rev apTs) (rev (map Init pTs))). G ⊢ x ≼i y"
      by (auto simp add: zip_rev)
    with wfprog l l_o opTs
    have "approx_loc G hp ihp opTs (map OK (rev (map Init pTs)))"
      by (auto intro: assConv_approx_stk_imp_approx_loc)
    hence "approx_stk G hp ihp opTs (rev (map Init pTs))"
      by (simp add: approx_stk_def)
    hence "approx_stk G hp ihp (rev opTs) (map Init pTs)"
      by (simp add: approx_stk_rev)
    hence "approx_loc G hp ihp (rev opTs) (map OK (map Init pTs))"
      by (simp add: approx_stk_def)
    with new_Addr new_ref'
    have "approx_loc G hp ?ihp' (rev opTs) (map OK (map Init pTs))"
      by (auto dest: approx_loc_newref)
    moreover
    from new_Addr new_ref' have "hp ≤| ?hp'" by simp
    ultimately
    have "approx_loc G ?hp' ?ihp' (rev opTs) (map OK (map Init pTs))"
      by (rule approx_loc_imp_approx_loc_sup_heap)
    with r a l_o l
    have "approx_loc G ?hp' ?ihp' ?loc' ?LT"
      by (auto simp add: approx_loc_append)
    from wfprog this sup_loc
    have loc: "approx_loc G ?hp' ?ihp' ?loc' LT0"
      by (rule approx_loc_imp_approx_loc_sup)
```

```
from new_Addr new_ref' l_o l have
  "corresponds [] ?loc' ([], ?LT) ?ihp' (Addr ref') (PartInit C')"
  apply (simp add: corresponds_def corr_stk_def corr_loc_cons
              split del: split_if)
  apply (rule conjI)
   apply (simp add: corr_val_def)
  apply (rule corr_loc_start)
    apply clarsimp
    apply (erule disjE)
     apply (erule imageE, erule imageE)
     apply simp
     apply blast
    apply (drule in_set_replicateD)
    apply assumption
   apply simp
  apply blast
  done

with sup_loc have corr':
  "corresponds [] ?loc' ([], LT0) ?ihp' (Addr ref') (PartInit C')"
  by (fastsimp elim: corresponds_widen_split)

from l_o l
have "length (rev opTs @ replicate mxl' arbitrary) =
      length (map OK (map Init pTs) @ replicate mxl' Err)" by simp
moreover
have "∀x∈set (map OK (map Init pTs) @ replicate mxl' Err).
      x = Err ∨ (∃t. x = OK (Init t))"
  by (auto dest: in_set_replicateD)
ultimately
have "consistent_init [] ?loc' ([], ?LT) ?ihp'"
  apply (unfold consistent_init_def)
  apply (unfold corresponds_def)
  apply (simp (no_asm) add: corr_stk_def corr_loc_empty corr_loc_cons
                split del: split_if)
  apply safe
   apply (rule exI)
   apply (rule conjI)
    apply (simp (no_asm))
   apply (rule corr_loc_start)
    apply assumption+
    apply blast
   apply (rule exI)
   apply (rule conjI)
    defer
    apply (blast intro: corr_loc_start)
   apply (rule impI)
   apply (simp add: new_Addr split del: split_if)
```

```
    apply (rule conjI)
     apply (rule refl)
    apply (simp add: corr_val_def new_Addr split: split_if_asm)
    done
  with sup_loc have "consistent_init [] ?loc' ([], LT0) ?ihp'"
    by (auto intro: consistent_init_widen_split)

  with start loc l_o l corr' new_Addr new_ref'
  show ?thesis by (simp add: correct_frame_def split: split_if_asm)
qed

from a_stk a_loc init suc_l pc new_Addr new_ref' s corr
have c_f': "correct_frame G ?hp' ?ihp' (rev apTs @ X # ST, LT) maxl ins ?f'"
  apply (unfold correct_frame_def)
  apply simp
  apply (rule conjI)
   apply (rule impI)
   apply (rule conjI)
    apply (drule approx_stk_newref, assumption)
    apply (rule approx_stk_imp_approx_stk_sup_heap, assumption)
    apply simp
   apply (rule conjI)
    apply (drule approx_loc_newref, assumption)
    apply (rule approx_loc_imp_approx_loc_sup_heap, assumption)
    apply simp
   apply (rule conjI)
    apply (rule consistent_init_new_val, assumption+)
    apply (rule impI, erule impE, assumption, elim exE conjE)
    apply (rule conjI)
     apply (rule corresponds_new_val2, assumption+)
     apply simp
     apply (rule exI, rule conjI)
       apply (rule impI)
       apply assumption
      apply simp
     apply (rule impI)
     apply (rule conjI)
      apply (drule approx_stk_newref, assumption)
      apply (rule approx_stk_imp_approx_stk_sup_heap, assumption)
      apply simp
     apply (rule conjI)
      apply (drule approx_loc_newref, assumption)
      apply (rule approx_loc_imp_approx_loc_sup_heap, assumption)
      apply simp
     apply (rule conjI)
      apply (rule consistent_init_new_val, assumption+)
     apply (rule impI, erule impE, assumption, elim exE conjE)
     apply (rule conjI)
```

```
      apply (rule corresponds_new_val2, assumption+)
     apply simp
     apply (rule exI, rule conjI)
      apply (rule impI)
      apply (rule conjI)
       apply assumption
      apply simp
     apply simp
     done

from new_Addr new_ref' heap_ok is_class_D wfprog
have hp'_ok: "G ⊢h ?hp' √"
  by (auto intro: hconf_imp_hconf_newref oconf_blank)

with new_Addr new_ref'
have ihp'_ok: "h_init G ?hp' ?ihp'"
  by (auto intro!: h_init_newref)

from hp_Some new_ref' have neq_ref: "ref ≠ ref'" by auto

with new_Addr oX_Addr X oX hp_Some
have ctor_ok:
  "constructor_ok G ?hp' ?ihp' (Addr ref) C' (C'=Object) ?r'"
  apply (unfold constructor_ok_def)
  apply (simp add: approx_val_def iconf_def)
  apply (erule disjE)
   apply (clarsimp simp add: blank_def)
  apply (elim exE conjE)
  apply (simp add: blank_def)
  done

from new_Addr new_ref' frames
have c_frs: "correct_frames G ?hp' ?ihp' phi rT sig z r frs"
  by (auto simp add: blank_def
           intro: correct_frames_imp_correct_frames_newref)

from new_Addr new_ref' prealloc have prealloc': "preallocated ?hp' ?ihp'"
  by (auto intro: preallocated_newref)

from state'_val heap_ok mC' ins method phi_pc s l ctor_ok c_f' c_frs
     frames s' LT0 c_f is_class_C is_class new_Addr new_ref' hp'_ok ihp'_ok
     prealloc'
show "G,phi ⊢JVM state' √"
  apply (unfold correct_state_def)
  apply (simp split del: split_if)

  apply (intro exI conjI impI)
     apply (rule refl)
```

```
     apply assumption
    apply (simp add: oX_pos oX_Addr hp_Some neq_ref)
    done
qed

lemmas [simp del] = map_append

lemma Return_correct_not_init:
"⟦ fst sig ≠ init;
  wt_jvm_prog G phi;
  method (G,C) sig = Some (C,rT,maxs,maxl,ins,et);
  ins ! pc = Return;
  wt_instr (ins!pc) G C rT (phi C sig) maxs (fst sig = init) (length ins) et pc;
  Some state' = exec (G, None, hp, ihp, (stk,loc,C,sig,pc,r)#frs) ;
  G,phi ⊢JVM (None, hp, ihp, (stk,loc,C,sig,pc,r)#frs)√ ⟧
⟹ G,phi ⊢JVM state'√"
proof -
  assume wtjvm: "wt_jvm_prog G phi"
  assume mthd:  "method (G,C) sig = Some (C,rT,maxs,maxl,ins,et)"
  assume ins:   "ins!pc = Return"
  assume wt:    "wt_instr (ins!pc) G C rT (phi C sig) maxs
                          (fst sig = init) (length ins) et pc"
  assume state: "Some state' = exec (G, None, hp, ihp, (stk,loc,C,sig,pc,r)#frs)"
  assume approx:"G,phi ⊢JVM (None, hp, ihp, (stk,loc,C,sig,pc,r)#frs)√"

  from approx mthd
  obtain s z where
    heap_ok: "G ⊢h hp√" and
    init_ok: "h_init G hp ihp" and
    prealloc:"preallocated hp ihp" and
    classC:  "is_class G C" and
    some:    "phi C sig ! pc = Some (s, z)" and
    frame:   "correct_frame G hp ihp s maxl ins (stk, loc, C, sig, pc, r)" and
    frames:  "correct_frames G hp ihp phi rT sig z r frs"
    ..

  obtain mn pTs where sig: "sig = (mn,pTs)" by (cases sig)
  moreover assume "fst sig ≠ init"
  ultimately have not_init: "mn ≠ init" by simp

  from ins some sig wt
  obtain T ST LT where
    s:  "s = (T # ST, LT)" and
    T:  "G ⊢ T ≼i Init rT" and
    pc: "pc < length ins"
    by (simp add: wt_instr_def eff_def eff_bool_def norm_eff_def) blast

  from frame s
```

```
obtain rval stk' loc where
  stk: "stk = rval # stk'" and
  val: "approx_val G hp ihp rval (OK T)" and
  approx_stk: "approx_stk G hp ihp stk' ST" and
  approx_val: "approx_loc G hp ihp loc LT" and
  consistent: "consistent_init stk loc (T # ST, LT) ihp" and
  len: "length loc = Suc (length (snd sig) + maxl)"
  by (simp add: correct_frame_def) blast

from stk mthd ins state
have "frs = [] ⟹ ?thesis" by (simp add: correct_state_def)
moreover
{ fix f frs' assume frs: "frs = f#frs'"
  obtain stk0 loc0 C0 sig0 pc0 r0 where
    f: "f = (stk0, loc0, C0, sig0, pc0, r0)" by (cases f)

  let ?r'   = "(stk0 ! length pTs, snd r0)"
  let ?stk' = "drop (Suc (length pTs)) stk0"
  let ?f'   = "(rval#?stk',loc0,C0,sig0,Suc pc0,r0)"

  from stk mthd ins sig f frs not_init state
  have state': "state' = Norm (hp, ihp, ?f' # frs')" by (simp add: split_beta)

  from f frs frames sig
  obtain ST0 LT0 T0 z0 rT0 maxs0 maxl0 ins0 et0 C' apTs D D' rT' body' where
    class_C0: "is_class G C0" and
    methd_C0: "method (G, C0) sig0 = Some (C0, rT0, maxs0, maxl0, ins0, et0)" and
    ins0:     "ins0 ! pc0 = Invoke C' mn pTs ∨
               ins0 ! pc0 = Invoke_special C' mn pTs" and
    phi_pc0:  "phi C0 sig0 ! pc0 = Some ((rev apTs @ T0 # ST0, LT0), z0)" and
    apTs:     "length apTs = length pTs" and
    methd_C': "method (G, C') sig = Some (D', rT', body')" "G ⊢ rT ≼ rT'" and
    c_fr:     "correct_frame G hp ihp (rev apTs @ T0 # ST0, LT0) maxl0 ins0
                          (stk0, loc0, C0, sig0, pc0, r0)" and
    c_frs:    "correct_frames G hp ihp phi rT0 sig0 z0 r0 frs'"
    apply simp
    apply (elim conjE exE)
    apply (rule that)
     apply assumption+
     apply simp
     apply ( (rule conjI,rule refl)+ )
     apply (rule refl)
     apply assumption+
     apply simp
     apply assumption
    done

  from c_fr obtain
```

```
    a_stk0: "approx_stk G hp ihp stk0 (rev apTs @ T0 # ST0)" and
    a_loc0: "approx_loc G hp ihp loc0 LT0" and
    cons0:  "consistent_init stk0 loc0 (rev apTs @ T0 # ST0, LT0) ihp" and
    corr0:  "fst sig0 = init ⟶
    corresponds stk0 loc0 (rev apTs @ T0 # ST0, LT0) ihp (fst r0) (PartInit C0) ∧
    (∃l. fst r0 = Addr l ∧ hp l ≠ None ∧
    (ihp l = PartInit C0 ∨ (∃C'. ihp l = Init (Class C'))))" and
    pc0:    "pc0 < length ins0" and
    lenloc0:"length loc0 = Suc (length (snd sig0) + maxl0)"
    by (unfold correct_frame_def) simp

  from pc0 wtjvm methd_C0 have wt0:
    "wt_instr (ins0!pc0) G C0 rT0 (phi C0 sig0) maxs0
             (fst sig0 = init) (length ins0) et0 pc0"
    by - (rule wt_jvm_prog_impl_wt_instr)

  from ins0 apTs phi_pc0 not_init sig methd_C' wt0
  obtain ST'' LT'' where
    Suc_pc0:     "Suc pc0 < length ins0" and
    phi_Suc_pc0: "phi C0 sig0 ! Suc pc0 = Some ((ST'',LT''),z0)" and
    T0:          "G ⊢ T0 ≼i Init (Class C')" and
    less:        "G ⊢ (Init rT' # ST0, LT0) <=s (ST'',LT'')"
    apply (simp add: wt_instr_def)
    apply (erule disjE)
     apply (simp add: eff_def eff_bool_def norm_eff_def)
     apply (elim exE conjE)
     apply (rotate_tac -7)
     apply clarsimp
     apply blast
    apply simp
    done

  from wtjvm obtain mb where wf: "wf_prog mb G" by (simp add: wt_jvm_prog_def)

  from a_stk0 obtain apts v stk0' where
    stk0':   "stk0 = apts @ v # stk0'" and
    len:     "length apts = length apTs" and
    v:       "approx_val G hp ihp v (OK T0)" and
    a_stk0': "approx_stk G hp ihp stk0' ST0"
    by - (drule approx_stk_append_lemma, auto)

  from stk0' len v a_stk0' wf apTs val T wf methd_C'
  have a_stk':
    "approx_stk G hp ihp (rval # drop (Suc (length pTs)) stk0) (Init rT'#ST0)"
    apply simp
    apply (rule approx_val_widen, assumption+)
    apply (clarsimp simp add: init_le_Init2)
    apply (erule widen_trans)
```

```
  apply assumption
  done

from stk0' len apTs cons0
have cons':
  "consistent_init (rval # drop (Suc (length pTs)) stk0) loc0
                  (Init rT'#ST0, LT0) ihp"
  apply simp
  apply (drule consistent_init_append)
   apply simp
  apply (drule consistent_init_pop)
  apply (rule consistent_init_Init_stk)
  apply assumption
  done

from stk0' len apTs corr0
have corr':
  "fst sig0 = init ⟶
  corresponds (rval # drop (Suc (length pTs)) stk0) loc0
             (Init rT'#ST0, LT0) ihp (fst r0) (PartInit C0) ∧
  (∃l. fst r0 = Addr l ∧ hp l ≠ None ∧
      (ihp l = PartInit C0 ∨ (∃C'. ihp l = Init (Class C'))))"
  apply clarsimp
  apply (drule corresponds_append)
   apply simp
  apply (simp add: corresponds_stk_cons)
  done

with cons' lenloc0 a_loc0 Suc_pc0 a_stk' wf
have frame':
  "correct_frame G hp ihp (ST'',LT'') maxl0 ins0 ?f'"
  apply (simp add: correct_frame_def)
  apply (insert less)
  apply (clarsimp simp add: sup_state_conv map_eq_Cons)
  apply (rule conjI)
   apply (rule approx_val_widen, assumption+)
  apply (rule conjI)
   apply (rule approx_stk_imp_approx_stk_sup, assumption+)
  apply (rule conjI)
   apply (rule approx_loc_imp_approx_loc_sup, assumption+)
  apply (rule conjI)
   apply (rule consistent_init_widen_split, assumption)
    apply simp
    apply blast
   apply assumption
  apply (rule impI, erule impE, assumption, erule conjE)
  apply (rule corresponds_widen_split, assumption)
    apply simp
```

```
        apply blast
       apply assumption
      apply blast
      done

    from state' heap_ok init_ok frame' c_frs class_C0 methd_C0 phi_Suc_pc0 prealloc
    have ?thesis by (simp add: correct_state_def)
  }
  ultimately
  show "G,phi ⊢JVM state'√" by (cases frs, blast+)
qed



lemma Return_correct_init:
"⟦ fst sig = init;
  wt_jvm_prog G phi;
  method (G,C) sig = Some (C,rT,maxs,maxl,ins,et);
  ins ! pc = Return;
  wt_instr (ins!pc) G C rT (phi C sig) maxs (fst sig = init) (length ins) et pc;
  Some state' = exec (G, None, hp, ihp, (stk,loc,C,sig,pc,r)#frs);
  G,phi ⊢JVM (None, hp, ihp, (stk,loc,C,sig,pc,r)#frs)√ ⟧
⟹ G,phi ⊢JVM state'√"
proof -
  assume wtjvm: "wt_jvm_prog G phi"
  assume mthd:  "method (G,C) sig = Some (C,rT,maxs,maxl,ins,et)"
  assume ins:   "ins!pc = Return"
  assume wt:    "wt_instr (ins!pc) G C rT (phi C sig) maxs
                          (fst sig = init) (length ins) et pc"
  assume state: "Some state' = exec (G, None, hp, ihp, (stk,loc,C,sig,pc,r)#frs)"
  assume approx:"G,phi ⊢JVM (None, hp, ihp, (stk,loc,C,sig,pc,r)#frs)√"

  from approx mthd
  obtain s z where
    heap_ok: "G ⊢h hp√" and
    init_ok: "h_init G hp ihp" and
    prealloc:"preallocated hp ihp" and
    classC:  "is_class G C" and
    some:    "phi C sig ! pc = Some (s, z)" and
    frame:   "correct_frame G hp ihp s maxl ins (stk, loc, C, sig, pc, r)" and
    frames:  "correct_frames G hp ihp phi rT sig z r frs"
    ..

  obtain mn pTs where sig: "sig = (mn,pTs)" by (cases sig)
  moreover assume sig_in: "fst sig = init"
  ultimately have is_init: "mn = init" by simp

  from ins some sig is_init wt
```

```
obtain T ST LT where
  s:  "s = (T # ST, LT)" and
  T:  "G ⊢ T ≼i Init rT" and
  z:  "z" and
  pc: "pc < length ins"
  by (simp add: wt_instr_def eff_def eff_bool_def norm_eff_def) blast

from frame s sig_in
obtain rval stk' loc where
  stk: "stk = rval # stk'" and
  val: "approx_val G hp ihp rval (OK T)" and
  approx_stk: "approx_stk G hp ihp stk' ST" and
  approx_val: "approx_loc G hp ihp loc LT" and
  consistent: "consistent_init stk loc (T # ST, LT) ihp" and
  corr: "corresponds stk loc (T # ST, LT) ihp (fst r) (PartInit C)" and
  len: "length loc = Suc (length (snd sig) + maxl)"
  by (simp add: correct_frame_def) blast

from stk mthd ins state
have "frs = [] ⟹ ?thesis" by (simp add: correct_state_def)
moreover
{ fix f frs' assume frs: "frs = f#frs'"
  obtain stk0 loc0 C0 sig0 pc0 r0 where
    f: "f = (stk0, loc0, C0, sig0, pc0, r0)" by (cases f)

  from f frs frames sig is_init
  obtain ST0 LT0 T0 z0 rT0 maxs0 maxl0 ins0 et0 C' apTs D D' rT' body' where
    class_C0: "is_class G C0" and
    methd_C0: "method (G, C0) sig0 = Some (C0, rT0, maxs0, maxl0, ins0, et0)" and
    ins0:     "ins0 ! pc0 = Invoke C' mn pTs ∨
               ins0 ! pc0 = Invoke_special C' mn pTs" and
    phi_pc0:  "phi C0 sig0 ! pc0 = Some ((rev apTs @ T0 # ST0, LT0), z0)" and
    apTs:     "length apTs = length pTs" and
    methd_C': "method (G, C') sig = Some (D', rT', body')" "G ⊢ rT ≼ rT'" and
    ctor_ok:  "constructor_ok G hp ihp (stk0 ! length apTs) C' z r" and
    c_fr:     "correct_frame G hp ihp (rev apTs @ T0 # ST0, LT0) maxl0 ins0
                             (stk0, loc0, C0, sig0, pc0, r0)" and
    c_frs:    "correct_frames G hp ihp phi rT0 sig0 z0 r0 frs'"
    apply simp
    apply (elim conjE exE)
    apply (rule that)
     apply assumption+
     apply simp
     apply assumption+
     apply simp
     apply ( (rule conjI,rule refl)+ )
     apply (rule refl)
     apply assumption+
```

```
    apply simp
    apply assumption
   done

  from c_fr
  obtain
    a_stk0: "approx_stk G hp ihp stk0 (rev apTs @ T0 # ST0)" and
    a_loc0: "approx_loc G hp ihp loc0 LT0" and
    cons0:  "consistent_init stk0 loc0 (rev apTs @ T0 # ST0, LT0) ihp" and
    corr0:  "fst sig0 = init ⟶
    corresponds stk0 loc0 (rev apTs @ T0 # ST0, LT0) ihp (fst r0) (PartInit C0) ∧
    (∃l. fst r0 = Addr l ∧ hp l ≠ None ∧
     (ihp l = PartInit C0 ∨ (∃C'. ihp l = Init (Class C'))))" and
    pc0:    "pc0 < length ins0" and
    lenloc0:"length loc0 = Suc (length (snd sig0) + maxl0)"
    by (unfold correct_frame_def) simp

  from pc0 wtjvm methd_C0 have wt0:
    "wt_instr (ins0!pc0) G C0 rT0 (phi C0 sig0) maxs0
              (fst sig0 = init) (length ins0) et0 pc0"
    by - (rule wt_jvm_prog_impl_wt_instr)

  let ?z'  = "if ∃D. T0 = PartInit D then True else z0"
  let ?ST' = "Init rT' # replace T0 (Init (theClass T0)) ST0"
  let ?LT' = "replace (OK T0) (OK (Init (theClass T0))) LT0"

  from ins0 apTs phi_pc0 is_init sig methd_C' wt0
  obtain ST'' LT'' where
    Suc_pc0: "Suc pc0 < length ins0" and
    phi_Suc_pc0: "phi C0 sig0 ! Suc pc0 = Some ((ST'',LT''),?z')" and
    T0:      "(∃pc. T0 = UnInit C' pc) ∨
              (T0 = PartInit C0 ∧ G ⊢ C0 ≺C1 C' ∧ ¬z0)" and
    less:    "G ⊢ (?ST', ?LT') <=s (ST'',LT'')"
    apply (simp add: wt_instr_def)
    apply (erule disjE)
     apply simp
    apply (simp add: eff_def eff_bool_def norm_eff_def)
    apply (elim exE conjE)
    apply (clarsimp simp add: nth_append)
    apply blast
    done

  from a_stk0 obtain apts oX stk0' where
    stk0': "stk0 = apts @ oX # stk0'" "length apts = length apTs" and
    a_val: "approx_val G hp ihp oX (OK T0)" and
    a_stk: "approx_stk G hp ihp stk0' ST0"
    by - (drule approx_stk_append_lemma, auto)
```

```
with apTs have oX_pos: "stk0!length pTs = oX" by (simp add: nth_append)

let ?c    = "snd r"
let ?r'   = "if r0 = (oX,Null) then (oX, ?c) else r0"
let ?stk' = "rval#(replace oX ?c stk0')"
let ?loc' = "replace oX ?c loc0"
let ?f'   = "(?stk',?loc',C0,sig0,Suc pc0,?r')"

from stk apTs stk0' mthd ins sig f frs is_init state oX_pos
have state': "state' = Norm (hp, ihp, ?f' # frs')" by (simp add: split_beta)

from wtjvm obtain mb where "wf_prog mb G" by (simp add: wt_jvm_prog_def)

from ctor_ok z apTs oX_pos a_val T0
obtain C'' D pc' a c fs1 fs3 where
  a:     "oX = Addr a" and
  ihp_a: "ihp a = UnInit C'' pc' ∨ ihp a = PartInit D" and
  hp_a:  "hp a = Some (C'', fs1)" and
  c:     "snd r = Addr c" and
  ihp_c: "ihp c = Init (Class C'')" and
  hp_c:  "hp c = Some (C'', fs3)"
  by (simp add: constructor_ok_def, clarify) auto

from a_val a hp_a T0
have "G ⊢ C'' ≼C C'∧ (T0 = PartInit C0 ⟶ G ⊢ C'' ≼C C0)"
  apply -
  apply (erule disjE)
   apply (clarsimp simp add: approx_val_def iconf_def)
  apply (clarsimp simp add: approx_val_def iconf_def)
  apply (simp add: conf_def)
  apply (rule rtrancl_trans, assumption)
  apply blast
  done

with c hp_c ihp_c heap_ok T0
have a_val':
  "approx_val G hp ihp (snd r) (OK (Init (theClass T0)))"
  apply (simp add: approx_val_def iconf_def is_init_def)
  apply (erule disjE)
   apply clarsimp
   apply (rule conf_obj_AddrI, assumption+)
  apply clarsimp
  apply (rule conf_obj_AddrI, assumption+)
  done

from stk0' cons0  T0
have corr:
  "corresponds stk0' loc0 (ST0,LT0) ihp oX T0"
```

```
  apply simp
  apply (erule disjE)
   apply (elim exE)
   apply (drule consistent_init_append)
    apply simp
   apply (drule consistent_init_corresponds_stk_cons)
    apply blast
   apply (simp add: corresponds_stk_cons)
  apply (drule consistent_init_append)
   apply simp
  apply (drule consistent_init_corresponds_stk_cons)
   apply blast
  apply (simp add: corresponds_stk_cons)
  done

from a_val a_val' a_loc0 T0 corr
have a_loc':
  "approx_loc G hp ihp ?loc' ?LT'"
  by (auto elim: approx_loc_replace simp add: corresponds_def)

from wf T methd_C' a_val a_val' a_stk corr T0 val
have a_stk':
  "approx_stk G hp ihp ?stk' ?ST'"
  apply -
  apply clarsimp
  apply (rule conjI)
   apply (clarsimp simp add: approx_val_def iconf_def init_le_Init2)
   apply (rule conf_widen, assumption+)
   apply (erule widen_trans, assumption)
  apply (unfold approx_stk_def)
  apply (erule disjE)
   apply (elim exE conjE)
   apply (drule approx_loc_replace)
       apply assumption back
      apply assumption
     apply blast
    apply (simp add: corresponds_def corr_stk_def)
   apply (simp add: replace_map_OK)
  apply (elim exE conjE)
  apply (drule approx_loc_replace)
      apply assumption back
     apply assumption
    apply blast
   apply (simp add: corresponds_def corr_stk_def)
  apply (simp add: replace_map_OK)
  done

from stk0' apTs cons0
```

```
have "consistent_init stk0' loc0 (ST0,LT0) ihp"
  apply simp
  apply (drule consistent_init_append, simp)
  apply (erule consistent_init_pop)
  done

with a_stk a_loc0 a_val
have "consistent_init ?stk' ?loc' (?ST', ?LT') ihp"
  apply -
  apply (rule consistent_init_Init_stk)
  apply (erule consisten_init_replace)
   apply assumption+
  apply (rule refl)
  done
from this less
have cons':
  "consistent_init ?stk' ?loc' (ST'',LT'') ihp"
  by (rule consistent_init_widen)

from stk0' len apTs corr0
have
  "fst sig0 = init ⟶
  corresponds (rval # stk0') loc0 (Init rT'#ST0, LT0) ihp (fst r0) (PartInit C0)"
  apply clarsimp
  apply (drule corresponds_append)
   apply simp
  apply (simp add: corresponds_stk_cons)
  done
with a_stk a_loc0 a_val
have
  "fst sig0 = init ⟶
  corresponds ?stk' ?loc' (?ST', ?LT') ihp (fst r0) (PartInit C0)"
  apply -
  apply clarify
  apply (simp add: corresponds_stk_cons)
  apply (erule corresponds_replace)
    apply assumption+
   apply simp
  apply blast
  done
with less
have corr':
  "fst sig0 = init ⟶
  corresponds ?stk' ?loc' (ST'', LT'') ihp (fst r0) (PartInit C0)"
  apply -
  apply (rule impI, erule impE, assumption)
  apply (rule corresponds_widen)
  apply assumption+
```

```
    apply blast
    done

  have fst_r': "fst ?r' = fst r0" by simp

  from lenloc0 a_loc' Suc_pc0 a_stk' cons' less fst_r' corr' corr0
  have c_fr': "correct_frame G hp ihp (ST'', LT'') maxl0 ins0 ?f'"
    apply (unfold correct_frame_def)
    apply clarify
    apply (clarsimp simp add: sup_state_conv map_eq_Cons split del: split_if)
    apply (rule conjI)
     apply (rule approx_val_widen, assumption+)
    apply (rule conjI)
     apply (rule approx_stk_imp_approx_stk_sup, assumption+)
    apply (rule approx_loc_imp_approx_loc_sup, assumption+)
    done

  from c_frs
  have frs'1:
    "fst sig0 ≠ init ⟹ correct_frames G hp ihp phi rT0 sig0 ?z' ?r' frs'"
    apply -
    apply (cases frs')
     apply simp
    apply (clarsimp split del: split_if)
    apply (intro exI conjI)
        apply assumption
       apply (rule refl)
      apply assumption
     apply assumption
    apply assumption
    done
  moreover
  have frs'2: "frs' = [] ⟹ correct_frames G hp ihp phi rT0 sig0 ?z' ?r' frs'"
    by simp
  moreover
  { fix f'' frs''
    assume cons: "frs' = f'' # frs''"
    assume sig0_in: "fst sig0 = init"

    { assume eq: "oX = fst r0"
      with corr0 sig0_in
      have "corresponds stk0 loc0 (rev apTs @ T0 # ST0, LT0) ihp oX (PartInit C0)"
        by simp
      with stk0'
      have "corresponds (oX#stk0') loc0 (T0#ST0, LT0) ihp oX (PartInit C0)"
        apply simp
        apply (rule corresponds_append)
        apply assumption
```

```
      apply simp
      done
    with a ihp_a eq corr0 sig0_in T0
    have "ihp a = PartInit C0"
      by (clarsimp simp add: corresponds_stk_cons corr_val_def)
    with a T0 a_val
    have z0: "T0 = PartInit C0 ∧ ¬z0"
      by (auto simp add: approx_val_def iconf_def)
    from c_frs sig0_in z0 cons
    have "snd r0 = Null"
      apply -
      apply (drule correct_frames_ctor_ok)
      apply clarsimp
      apply (simp add: constructor_ok_def split_beta)
      done
    moreover
    have "r0 = (fst r0, snd r0)" by simp
    ultimately
    have eq_r0: "r0 = (oX, Null)" by (simp  add: eq)
    with apTs oX_pos ctor_ok c_frs z cons
    have "correct_frames G hp ihp phi rT0 sig0 True (oX, ?c) frs'"
      apply (clarsimp split del: split_if)
      apply (intro exI conjI)
           apply assumption
          apply (rule refl)
         apply assumption
        apply assumption
       apply assumption
      apply (rule impI, erule impE, assumption)
       apply (rule constructor_ok_pass_val)
         apply assumption
        apply simp
       apply simp
      done
    with oX_pos apTs z0 eq_r0
    have "correct_frames G hp ihp phi rT0 sig0 ?z' ?r' frs'" by simp
  }
  moreover
  { assume neq: "oX ≠ fst r0"
    with T0 stk0' corr0 sig0_in
    have "∃pc'. T0 = UnInit C' pc'"
      apply simp
      apply (erule disjE)
       apply simp
      apply clarify
      apply (drule corresponds_append)
       apply simp
      apply (simp add: corresponds_stk_cons)
```

```
        done
      with c_frs apTs oX_pos neq
      have "correct_frames G hp ihp phi rT0 sig0 ?z' ?r' frs'" by clarsimp
    }
    ultimately
    have "correct_frames G hp ihp phi rT0 sig0 ?z' ?r' frs'" by blast
  }
  ultimately
  have "correct_frames G hp ihp phi rT0 sig0 ?z' ?r' frs'"
    by (cases frs', blast+)

  with state' heap_ok init_ok c_frs class_C0 methd_C0 phi_Suc_pc0 c_fr' prealloc
  have ?thesis by (unfold correct_state_def) simp
}
ultimately
show "G,phi ⊢JVM state'√" by (cases frs, blast+)
qed

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 C rT (phi C sig) maxs (fst sig = init) (length ins) et pc;
  Some state' = exec (G, None, hp, ihp, (stk,loc,C,sig,pc,r)#frs) ;
  G,phi ⊢JVM (None, hp, ihp, (stk,loc,C,sig,pc,r)#frs)√ ⟧
⟹ G,phi ⊢JVM state'√"
  apply (cases "fst sig = init")
   apply (rule Return_correct_init)
   apply assumption+
  apply (rule Return_correct_not_init)
   apply simp
  apply assumption+
  done

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 C rT (phi C sig) maxs (fst sig = init) (length ins) et pc;
  Some state' = exec (G, None, hp, ihp, (stk,loc,C,sig,pc,r)#frs) ;
  G,phi ⊢JVM (None, hp, ihp, (stk,loc,C,sig,pc,r)#frs)√ ⟧
⟹ G,phi ⊢JVM state'√"
apply (elim correctE, assumption)
apply (clarsimp simp add: defs2)
apply (fast intro: approx_loc_imp_approx_loc_sup
                   approx_stk_imp_approx_stk_sup
```

```
                    consistent_init_widen_split
                    corresponds_widen_split)
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 C rT (phi C sig) maxs (fst sig = init) (length ins) et pc;
  Some state' = exec (G, None, hp, ihp, (stk,loc,C,sig,pc,r)#frs) ;
  G,phi ⊢JVM (None, hp, ihp, (stk,loc,C,sig,pc,r)#frs)√ ⟧
⟹ G,phi ⊢JVM state'√"
apply (elim correctE, assumption)
apply (clarsimp simp add: defs2)
apply (rule conjI)
 apply (rule impI)
 apply simp
 apply (rule conjI)
  apply (rule approx_stk_imp_approx_stk_sup, assumption+)
 apply (rule conjI)
  apply (erule approx_loc_imp_approx_loc_sup, assumption+)
 apply (rule conjI)
  apply (drule consistent_init_pop)+
  apply (erule consistent_init_widen_split, assumption+)
 apply (rule impI, erule impE, assumption, erule conjE)
  apply (drule corresponds_pop)+
  apply (fast elim!: corresponds_widen_split)
apply (rule impI)
apply (rule conjI)
 apply (rule approx_stk_imp_approx_stk_sup, assumption+)
apply (rule conjI)
 apply (erule approx_loc_imp_approx_loc_sup, assumption+)
apply (rule conjI)
 apply (drule consistent_init_pop)+
 apply (erule consistent_init_widen_split, assumption+)
apply (rule impI, erule impE, assumption, erule conjE)
 apply (drule corresponds_pop)+
 apply (fast elim!: corresponds_widen_split)
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 C rT (phi C sig) maxs (fst sig = init) (length ins) et pc;
  Some state' = exec (G, None, hp, ihp, (stk,loc,C,sig,pc,r)#frs) ;
  G,phi ⊢JVM (None, hp, ihp, (stk,loc,C,sig,pc,r)#frs)√ ⟧
⟹ G,phi ⊢JVM state'√"
```

```
apply (elim correctE, assumption)
apply (clarsimp simp add: defs2)
apply (fast intro: approx_loc_imp_approx_loc_sup
                   approx_stk_imp_approx_stk_sup
            elim: consistent_init_widen_split
                  corresponds_widen_split
            dest: consistent_init_pop corresponds_pop)
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 C rT (phi C sig) maxs (fst sig = init) (length ins) et pc;
  Some state' = exec (G, None, hp, ihp, (stk,loc,C,sig,pc,r)#frs) ;
  G,phi ⊢JVM (None, hp, ihp, (stk,loc,C,sig,pc,r)#frs)√ ⟧
⟹ G,phi ⊢JVM state'√"
apply (elim correctE, assumption)
apply (clarsimp simp add: defs2 map_eq_Cons)
apply (rule conjI)
 apply (erule approx_val_imp_approx_val_sup, assumption+)
 apply simp
apply (rule conjI)
 apply (erule approx_val_imp_approx_val_sup, assumption+)
 apply simp
apply (rule conjI)
 apply (blast intro: approx_stk_imp_approx_stk_sup)
apply (rule conjI)
 apply (blast intro: approx_loc_imp_approx_loc_sup)
apply (rule conjI)
 apply (drule consistent_init_Dup)
 apply (rule consistent_init_widen_split)
   apply assumption
  prefer 2
  apply assumption
 apply simp
apply (rule impI, erule impE, assumption, erule conjE)
apply (drule corresponds_Dup)
apply (erule corresponds_widen_split)
  prefer 2
  apply assumption
 apply simp
apply blast
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 C rT (phi C sig) maxs (fst sig = init) (length ins) et pc;
  Some state' = exec (G, None, hp, ihp, (stk,loc,C,sig,pc,r)#frs) ;
  G,phi ⊢JVM (None, hp, ihp, (stk,loc,C,sig,pc,r)#frs)√ ⟧
⟹ G,phi ⊢JVM state'√"
apply (elim correctE, assumption)
apply (clarsimp simp add: defs2 map_eq_Cons)
apply (rule conjI)
 apply (erule approx_val_imp_approx_val_sup, assumption+)
 apply simp
apply (rule conjI)
 apply (erule approx_val_imp_approx_val_sup, assumption+)
 apply simp
apply (rule conjI)
 apply (erule approx_val_imp_approx_val_sup, assumption+)
 apply simp
apply (rule conjI)
 apply (blast intro: approx_stk_imp_approx_stk_sup)
apply (rule conjI)
 apply (blast intro: approx_loc_imp_approx_loc_sup)
apply (rule conjI)
 apply (drule consistent_init_Dup_x1)
 apply (rule consistent_init_widen_split)
   apply assumption
  prefer 2
  apply assumption
 apply simp
apply (rule impI, erule impE, assumption, erule conjE)
apply (drule corresponds_Dup_x1)
apply (erule corresponds_widen_split)
  prefer 2
  apply assumption
 apply simp
apply blast
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 C rT (phi C sig) maxs (fst sig = init) (length ins) et pc;
  Some state' = exec (G, None, hp, ihp, (stk,loc,C,sig,pc,r)#frs) ;
  G,phi ⊢JVM (None, hp, ihp, (stk,loc,C,sig,pc,r)#frs)√ ⟧
⟹ G,phi ⊢JVM state'√"
apply (elim correctE, assumption)
apply (clarsimp simp add: defs2 map_eq_Cons)
apply (rule conjI)
```

```
 apply (erule approx_val_imp_approx_val_sup, assumption+)
 apply simp
apply (rule conjI)
 apply (erule approx_val_imp_approx_val_sup, assumption+)
 apply simp
apply (rule conjI)
 apply (erule approx_val_imp_approx_val_sup, assumption+)
 apply simp
apply (rule conjI)
 apply (erule approx_val_imp_approx_val_sup, assumption+)
 apply simp
apply (rule conjI)
 apply (blast intro: approx_stk_imp_approx_stk_sup)
apply (rule conjI)
 apply (blast intro: approx_loc_imp_approx_loc_sup)
apply (rule conjI)
 apply (drule consistent_init_Dup_x2)
 apply (rule consistent_init_widen_split)
   apply assumption
  prefer 2
  apply assumption
 apply simp
apply (rule impI, erule impE, assumption, erule conjE)
apply (drule corresponds_Dup_x2)
apply (erule corresponds_widen_split)
  prefer 2
  apply assumption
 apply simp
apply blast
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 C rT (phi C sig) maxs (fst sig = init) (length ins) et pc;
  Some state' = exec (G, None, hp, ihp, (stk,loc,C,sig,pc,r)#frs) ;
  G,phi ⊢JVM (None, hp, ihp, (stk,loc,C,sig,pc,r)#frs)√ ⟧
⟹ G,phi ⊢JVM state'√"
apply (elim correctE, assumption)
apply (clarsimp simp add: defs2 map_eq_Cons)
apply (rule conjI)
 apply (erule approx_val_imp_approx_val_sup, assumption+)
 apply simp
apply (rule conjI)
 apply (erule approx_val_imp_approx_val_sup, assumption+)
 apply simp
```

```
apply (rule conjI)
 apply (blast intro: approx_stk_imp_approx_stk_sup)
apply (rule conjI)
 apply (blast intro: approx_loc_imp_approx_loc_sup)
apply (rule conjI)
 apply (drule consistent_init_Swap)
 apply (rule consistent_init_widen_split)
   apply assumption
  prefer 2
  apply assumption
 apply simp
apply (rule impI, erule impE, assumption, erule conjE)
apply (drule corresponds_Swap)
apply (erule corresponds_widen_split)
  prefer 2
  apply assumption
 apply simp
apply blast
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 C rT (phi C sig) maxs (fst sig = init) (length ins) et pc;
  Some state' = exec (G, None, hp, ihp, (stk,loc,C,sig,pc,r)#frs) ;
  G,phi ⊢JVM (None, hp, ihp, (stk,loc,C,sig,pc,r)#frs)√ ⟧
⟹ G,phi ⊢JVM state'√"
apply (elim correctE, assumption)
apply (clarsimp simp add: defs2 map_eq_Cons approx_val_def)
apply (clarsimp simp add: iconf_def init_le_Init conf_def)
apply (simp add: is_init_def)
apply (drule consistent_init_pop)+
apply (simp add: corresponds_stk_cons)
apply (blast intro: approx_stk_imp_approx_stk_sup
                   approx_val_imp_approx_val_sup
                   approx_loc_imp_approx_loc_sup
                   corresponds_widen_split
                   consistent_init_Init_stk
                   consistent_init_widen_split)
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, ihp, (stk,loc,C,sig,pc,r)#frs) ;
```

```
  G,phi ⊢JVM (None, hp, ihp, (stk,loc,C,sig,pc,r)#frs)√;
  fst (exec_instr (ins!pc) G hp ihp stk loc C sig pc r 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, ihp, (stk,loc,C,sig,pc,r)#frs);
  G,phi ⊢JVM (None, hp, ihp, (stk,loc,C,sig,pc,r)#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 ihp stk loc C sig pc r frs)")
defer
apply (erule xcpt_correct, assumption+)
apply (cases "ins!pc")
prefer 8
apply (rule Invoke_correct, assumption+)
prefer 8
apply (rule Invoke_special_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
```

4.23.4 Main

```
lemma correct_state_impl_Some_method:
  "G,phi ⊢JVM (None, hp, ihp, (stk,loc,C,sig,pc,r)#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 ihp 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,ihp,frs) = Some state')"
by (clarsimp simp add: neq_Nil_conv split_beta)

lemma L1:
  "⟦wt_jvm_prog G phi; G,phi ⊢JVM (xp,hp,ihp,frs)√; xp=None; frs≠[]⟧
  ⟹ ∃state'. exec(G,xp,hp,ihp,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,ihp,(stk,loc,C,sig,pc,r)#frs); G,phi ⊢JVM s0 √⟧
⟹ ∃ST LT z. (phi C sig) ! pc = Some ((ST,LT),z) ∧
    approx_stk G hp ihp stk ST ∧
    approx_loc G hp ihp loc LT"
apply (drule BV_correct)
apply assumption+
apply (simp add: correct_state_def correct_frame_def split_def
           split: option.splits)
apply (case_tac s)
apply simp
apply (case_tac s)
```

```
apply simp
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: 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")
  assumes wf: "wf_prog wf_mb Γ"
  shows hinit_start: "h_init Γ (start_heap Γ) start_iheap"
  apply (unfold h_init_def start_heap_def start_iheap_def)
  apply (auto simp add: blank_def o_init_def l_init_def
                        is_init_def split: split_if_asm)
  apply (auto simp add: init_vars_def map_of_map)
  done

lemma consistent_init_start:
  "G ⊢ OK (Init (Class C)) <=o X ⟹
  consistent_init [] (Null#replicate n arbitrary) ([], X#replicate n Err) hp"
  apply (induct n)
  apply (auto simp add: consistent_init_def corresponds_def corr_stk_def corr_loc_def)
  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) ⟹ m ≠ init
  ⟹ 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 (simp add: wt_jvm_prog_def hinit_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)
   defer
   apply (clarsimp simp add: sup_state_conv dest!: loc_widen_Err)
   apply (simp add: consistent_init_start)
  apply (auto simp add: sup_state_conv approx_val_def iconf_def
                        is_init_def subtype_def dest!: widen_RefT
              split: err.splits split_if_asm init_ty.split)
  done

theorem
  fixes G :: jvm_prog ("Γ") and Phi :: prog_type ("Φ")
```

```
  assumes welltyped: "wt_jvm_prog Γ Φ" and
          main: "is_class Γ C" "method (Γ,C) (m,[]) = Some (C, b)" "m ≠ init"
  shows typesafe:
  "G ⊢ start_state Γ C m -jvm→ s  ⟹  Γ,Φ ⊢JVM s √"
proof -
  from welltyped main
  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
```

4.24 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 isRef [simp]:
  "G,hp,ihp ⊢ v ::≼i Init (RefT T) ⟹ isRef v"
  by (cases v) (auto simp add: iconf_def conf_def isRef_def)

lemma isIntg [simp]:
  "G,hp,ihp ⊢ v ::≼i Init (PrimT Integer) ⟹ isIntg v"
  by (cases v) (auto simp add: iconf_def conf_def)

declare approx_loc_len [simp] approx_stk_len [simp]



lemma list_all2I:
  "∀(x,y) ∈ set (zip a b). P x y ⟹ length a = length b ⟹ list_all2 P a b"
  by (simp add: list_all2_def)
```

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

```
theorem
  assumes welltyped: "wt_jvm_prog G Phi" and conforms: "G,Phi ⊢JVM s √"
  shows no_type_error: "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 ihp frs where s [simp]: "s = (xcp, hp, ihp, 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 r frs' where
      xcp [simp]: "xcp = None" and
      frs [simp]: "frs = (stk,loc,C,sig,pc,r)#frs'"
      by (clarsimp simp add: neq_Nil_conv) fast
```

```
from conforms obtain  ST LT z 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), z)" and
  frame:  "correct_frame G hp ihp (ST,LT) maxl ins (stk,loc,C,sig,pc,r)" and
  frames: "correct_frames G hp ihp Phi rT sig z r frs'"
  by simp (rule correct_stateE)

from frame obtain
  stk: "approx_stk G hp ihp stk ST" and
  loc: "approx_loc G hp ihp loc LT" and
  init: "fst sig = init ⟶
   corresponds stk loc (ST, LT) ihp (fst r) (PartInit C) ∧
   (∃l. fst r = Addr l ∧ hp l ≠ None ∧
   (ihp l = PartInit C ∨ (∃C'. ihp l = Init (Class C'))))" and
  pc:  "pc < length ins" and
  len: "length loc = length (snd sig) + maxl + 1"
  by (rule correct_frameE)

note approx_val_def [simp]

from welltyped meth conforms
have "wt_instr (ins!pc) G C rT (Phi C sig) maxs (fst sig=init) (length ins) et pc"
  by simp (rule wt_jvm_prog_impl_wt_instr_cor)
then obtain
  app: "app (ins!pc) G C pc maxs rT (fst sig=init) et (Some ((ST,LT),z))" and
  pc': "∀pc' ∈ set (succs (ins!pc) pc). pc' < length ins"
  by (simp add: wt_instr_def phi eff_def) blast

with stk loc
have "check_instr (ins!pc) G hp ihp stk loc C sig pc r (length ins) frs'"
proof (cases "ins!pc")
  case (Getfield F C)
  with app stk loc 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,ihp ⊢ v ::≼i Init (Class C)"
    by clarsimp (blast dest: iconf_widen [OF _ _ wf])
  from conf have isRef: "isRef v" by simp
  moreover {
    assume "v ≠ Null" with conf isRef have
      "∃D vs. hp (the_Addr v) = Some (D, vs) ∧
              is_init hp ihp v ∧ G ⊢ D ≼C C"
      by (fastsimp simp add: iconf_def conf_def isRef_def2)
  }
  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 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,ihp ⊢ v ::≼i Init vT" and
      confr: "G,hp,ihp ⊢ ref ::≼i Init (Class C)"
      by clarsimp (blast dest: iconf_widen [OF _ _ wf])
    from confr have isRef: "isRef ref" by simp
    moreover
    from confv have "is_init hp ihp v" by (simp add: iconf_def)
    moreover {
      assume "ref ≠ Null" with confr isRef have
        "∃D vs. hp (the_Addr ref) = Some (D, vs)
               ∧ is_init hp ihp ref ∧ G ⊢ D ≼C C"
        by (fastsimp simp add: iconf_def conf_def isRef_def2)
    }
    ultimately show ?thesis using Putfield field class stk confv
      by (clarsimp simp add: iconf_def)
  next
    case (Invoke C mn ps)
    with stk app
    show ?thesis
      apply clarsimp
      apply (clarsimp dest!: approx_stk_append_lemma simp add: nth_append)
      apply (drule iconf_widen [OF _ _ wf], assumption)
      apply (clarsimp simp add: iconf_def)
      apply (drule non_npD, assumption)
      apply clarsimp
      apply (drule widen_methd [OF _ wf], assumption)
      apply (clarsimp simp add: approx_stk_rev [symmetric])
      apply (drule list_all2I, assumption)
      apply (unfold approx_stk_def approx_loc_def)
      apply (simp add: list_all2_approx)
      apply (drule list_all2_iconf_widen [OF wf], assumption+)
      done
  next
    case (Invoke_special C mn ps)
    with stk app
    show ?thesis
      apply clarsimp
      apply (clarsimp dest!: approx_stk_append_lemma simp add: nth_append)
      apply (erule disjE)
       apply (clarsimp simp add: iconf_def isRef_def)
```

```
         apply (clarsimp simp add: approx_stk_rev [symmetric])
         apply (drule list_all2I, assumption)
         apply (simp add: list_all2_approx approx_stk_def approx_loc_def)
         apply (drule list_all2_iconf_widen [OF wf], assumption+)
        apply (clarsimp simp add: iconf_def isRef_def)
        apply (clarsimp simp add: approx_stk_rev [symmetric])
        apply (drule list_all2I, assumption)
        apply (unfold approx_stk_def approx_loc_def)
        apply (simp add: list_all2_approx)
        apply (drule list_all2_iconf_widen [OF wf], assumption+)
        done
    next
      case Return with  stk app init meth frames
      show ?thesis
        apply clarsimp
        apply (drule iconf_widen [OF _ _ wf], assumption)
        apply (clarsimp simp add: iconf_def neq_Nil_conv
                        constructor_ok_def is_init_def 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 aggresive 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
```

4.25 Example Welltypings

theory *BVExample = JVMListExample + BVSpecTypeSafe:*

This theory shows type correctness of the example program in section 3.8 (p. 60) 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.

4.25.1 Setup

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

```
axioms
  distinct_classes [simp]: "list_nam ≠ test_nam"
  distinct_fields  [simp]: "val_nam ≠ next_nam"
  distinct_meth_list [simp]: "append_name ≠ init"
  distinct_meth_test [simp]: "makelist_name ≠ init"

declare
  distinct_classes [symmetric, simp]
  distinct_fields  [symmetric, simp]
  distinct_meth_list [symmetric, simp]
  distinct_meth_test [symmetric, simp]
```

Abbreviations for theorem sets 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 = JVMSystemClasses_def SystemClassC_defs
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, [],[Object_ctor])"
  by (simp add: class_def system_defs E_def)

lemma class_NullPointer [simp]:
  "class E (Xcpt NullPointer) = Some (Object, [], [Default_ctor])"
  by (simp add: class_def system_defs E_def)

lemma class_OutOfMemory [simp]:
  "class E (Xcpt OutOfMemory) = Some (Object, [], [Default_ctor])"
  by (simp add: class_def system_defs E_def)

lemma class_ClassCast [simp]:
  "class E (Xcpt ClassCast) = Some (Object, [], [Default_ctor])"
  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)

lemma class_test [simp]:
  "class E test_name = Some test_class"
  by (simp add: class_def system_defs E_def name_defs)

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)
  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) sig =
  (if sig = (init,[]) then
    Some (Object, PrimT Void, Suc 0, 0, [LitPush Unit, Return], [])
  else None)"
  by (simp add: method_rec_lemma [OF class_Object wf_subcls1_E] Object_ctor_def)

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 add: name_defs Default_ctor_def)
  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 add: name_defs Default_ctor_def)
  done

lemma method_default_ctor [simp]:
  "method (E, list_name) (init, []) =
  Some (list_name,PrimT Void,1,0,[Load 0,Invoke_special Object init [],Return],[])"
  apply (insert class_list)
  apply (unfold list_class_def)
  apply (drule method_rec_lemma [OF _ wf_subcls1_E])
  apply (simp add: name_defs Default_ctor_def)
  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
```

4.25.2 Program structure

The program is structurally wellformed:

```
lemma wf_struct:
  "wf_prog (λG C mb. True) E" (is "wf_prog ?mb E")
proof -
  note simps [simp] = wf_mdecl_def wf_mhead_def wf_cdecl_def system_defs
  have "unique E"
    by (simp add: E_def class_defs name_defs)
  moreover
  have "set JVMSystemClasses ⊆ set E" by (simp add: E_def)
  hence "wf_syscls E" by (rule wf_syscls)
  moreover
  have "wf_cdecl ?mb E ObjectC" by simp
  moreover
  have "wf_cdecl ?mb E NullPointerC"
    by (auto elim: notin_rtrancl simp add: name_defs subcls1)
  moreover
  have "wf_cdecl ?mb E ClassCastC"
    by (auto elim: notin_rtrancl simp add: name_defs subcls1)
  moreover
  have "wf_cdecl ?mb E OutOfMemoryC"
    by (auto elim: notin_rtrancl simp add: name_defs subcls1)
  moreover
  have "wf_cdecl ?mb E (list_name, list_class)"
    apply (auto elim!: notin_rtrancl
            simp add: wf_fdecl_def list_class_def subcls1)
    apply (auto simp add: name_defs)
    done
  moreover
  have "wf_cdecl ?mb E (test_name, test_class)"
    apply (auto elim!: notin_rtrancl
            simp add: wf_fdecl_def test_class_def subcls1)
    apply (auto simp add: name_defs)
    done
  ultimately
  show ?thesis by (simp del: simps add: wf_prog_def E_def JVMSystemClasses_def)
qed
```

4.25.3 Welltypings

We show welltypings of all methods in the program *E*. The more interesting ones are *append_name* in class *list_name*, and *makelist_name* in class *test_name*. The rest are default constructors.

```
lemmas eff_simps [simp] = eff_def norm_eff_def xcpt_eff_def eff_bool_def subtype_def
declare
  appInvoke [simp del]
  appInvoke_special [simp del]

constdefs
  phi_obj_ctor :: method_type ("φ_o")
  "φ_o ≡ [Some (([],                    [OK (Init (Class Object))]), True),
          Some (([Init (PrimT Void)], [OK (Init (Class Object))]), True)]"
```

4.25.4 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':
  "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 [simp]:
  "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 list_all' check_bounded_def)

declare list_all'_def [simp]


lemma init_tys_def2 [simp]:
  "init_tys E = {Init y | y. is_type E y} ∪ {UnInit c n | c n. True} ∪ {PartInit c |
c. True}"
  apply (unfold init_tys_def JType.esl_def)
  apply auto
  done


lemma check_types_lemma [simp]:
  "b ∈ list mxr (err (init_tys E)) ⟹
  a ∈ list (length a) (init_tys E) ⟹
```

```
  length a ≤ mxs ⟹
  OK (Some ((a, b), x)) ∈ states E mxs mxr"
  apply (unfold states_def)
  apply (unfold JVMType.sl_def)
  apply (unfold Err.sl_def Err.esl_def Opt.esl_def Product.esl_def
                stk_esl_def reg_sl_def TrivLat.esl_def upto_esl_def Listn.sl_def
                Init.esl_def)
  apply (simp (no_asm) del: init_tys_def2)
  apply blast
  done


lemma Object_init [simp]:
  "wt_method E Object init [] (PrimT Void) (Suc 0) 0 [LitPush Unit, Return] [] φo"
  apply (simp add: wt_method_def phi_obj_ctor_def
                   wt_start_def wt_instr_def)
  apply (simp add: check_types_def)
  apply clarify
  apply (elim pc_end pc_next pc_0)
  apply fastsimp
  apply fastsimp
  done

constdefs
  phi_default_ctor :: "cname ⇒ method_type" ("φc _")
  "φc C ≡ [Some (([],                  [OK (PartInit C)]), False),
           Some (([PartInit C],        [OK (PartInit C)]), False),
           Some (([Init (PrimT Void)], [OK (Init (Class C))]),  True)]"

lemma [simp]: "Ex Not" by fast
lemma [simp]: "∃z. z" by fast

lemma default_ctor [simp]:
  "E ⊢ C ≺C1 Object ⟹  is_class E C ⟹
  wt_method E C init [] (PrimT Void) (Suc 0) 0
           [Load 0, Invoke_special Object init [], Return] [] (φc C)"
  apply (simp add: wt_method_def phi_default_ctor_def
                   wt_start_def wt_instr_def check_types_def)
  apply (rule conjI)
  apply clarify
  apply (elim pc_end pc_next pc_0)
  apply simp
  apply (simp add: app_def xcpt_app_def replace_def)
  apply simp
  apply (fast dest: subcls1_wfD [OF _ wf_struct])
  done
```

constdefs

```
  phi_append :: method_type ("φa")
  "φa ≡ map (λ(x,y). Some ((map Init x, map (OK ∘ Init) y), False)) [
   (                                   [], [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 *wt_append [simp]:*

```
  "wt_method E list_name append_name [Class list_name] (PrimT Void) 3 0 append_ins
           [(Suc 0, 2, 8, Xcpt NullPointer)] φa"
  apply (simp add: wt_method_def append_ins_def phi_append_def
                   wt_start_def wt_instr_def name_defs)
  apply (fold name_defs)
  apply (simp add: check_types_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
  apply (simp add: match_exception_entry_def)
  apply simp
  apply simp
  done
```

Some abbreviations for readability

syntax

```
  list :: ty
  test :: ty
  intg :: ty
```

```
  void :: ty
translations
  "list" == "Init (Class list_name)"
  "test" == "Init (Class test_name)"
  "intg" == "Init (PrimT Integer)"
  "void" == "Init (PrimT Void)"

constdefs
  phi_makelist :: method_type ("φm")
  "φm ≡ map (λx. Some (x,False)) [
  (                                       [], [OK test,     Err,     Err]),
  (                      [UnInit list_name 0], [OK test,     Err,     Err]),
  (   [UnInit list_name 0, UnInit list_name 0], [OK test,     Err,     Err]),
  ([UnInit list_name 0,UnInit list_name 0,UnInit list_name 0],[OK test,Err,Err]),
  (                        [void, list, list], [OK test,     Err,     Err]),
  (                              [list, list], [OK test,     Err,     Err]),
  (                                    [list], [OK list,     Err,     Err]),
  (                              [intg, list], [OK list,     Err,     Err]),
  (                                        [], [OK list,     Err,     Err]),
  (                      [UnInit list_name 8], [OK list,     Err,     Err]),
  (   [UnInit list_name 8, UnInit list_name 8], [OK list,     Err,     Err]),
  (                              [void, list], [OK list,     Err,     Err]),
  (                                    [list], [OK list,     Err,     Err]),
  (                              [list, list], [OK list,     Err,     Err]),
  (                                    [list], [OK list, OK list,     Err]),
  (                              [intg, list], [OK list, OK list,     Err]),
  (                                        [], [OK list, OK list,     Err]),
  (                     [UnInit list_name 16], [OK list, OK list,     Err]),
  ( [UnInit list_name 16, UnInit list_name 16], [OK list, OK list,     Err]),
  (                              [void, list], [OK list, OK list,     Err]),
  (                                    [list], [OK list, OK list,     Err]),
  (                              [list, list], [OK list, OK list,     Err]),
  (                                    [list], [OK list, OK list, OK list]),
  (                              [intg, list], [OK list, OK list, OK list]),
  (                                        [], [OK list, OK list, OK list]),
  (                                    [list], [OK list, OK list, OK list]),
  (                              [list, list], [OK list, OK list, OK list]),
  (                                    [void], [OK list, OK list, OK list]),
  (                              [list, void], [OK list, OK list, OK list]),
  (                        [list, list, void], [OK list, OK list, OK list]),
  (                              [void, void], [OK list, OK list, OK list])
  ]"

lemma wt_makelist [simp]:
  "wt_method E test_name makelist_name [] (PrimT Void) 3 2 make_list_ins [] φm"
  apply (simp add: wt_method_def make_list_ins_def phi_makelist_def)
  apply (simp add: wt_start_def nat_number)
  apply (simp add: wt_instr_def name_defs)
```

```
apply (fold name_defs)
apply (simp add: check_types_def)
apply clarify
apply (elim pc_end pc_next pc_0)
apply (simp add: replace_def)
apply simp
apply simp
apply (simp add: app_def xcpt_app_def replace_def)
apply simp
apply simp
apply simp
apply simp
apply (simp add: replace_def)
apply simp
apply simp
apply (simp add: app_def xcpt_app_def replace_def)
apply simp
apply simp
apply simp
apply simp
apply simp
apply (simp add: replace_def)
apply simp
apply simp
apply (simp add: app_def xcpt_app_def replace_def)
apply simp
apply simp
apply simp
apply simp
apply simp
apply simp
apply simp
apply (simp add: app_def xcpt_app_def)
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 sig ≡ if C = Object ∧ sig = (init,[]) then φo else
             if C = test_name ∧ sig = (makelist_name, []) then φm else
             if C = list_name ∧ sig = (append_name, [Class list_name]) then φa else
             if sig = (init,[]) then φc C else []"
```

lemma *[simp]:*

```
  "is_class E C =
  (C ∈ {Cname list_nam, Cname test_nam, Xcpt NullPointer, Xcpt ClassCast, Xcpt OutOfMemory,
Object})"
  apply (insert E_classes)
  apply (auto simp add: name_defs)
  done

declare is_class_def [simp del]

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)
  apply auto
  apply (auto simp add: Phi_def)
  apply (insert subcls1)
  apply (auto simp add: name_defs)
  done
```

4.25.5 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 (auto simp add: is_class_def)
  done

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.