

μ Java

Gerwin Klein
Tobias Nipkow
David von Oheimb
Cornelia Pusch

October 11, 2000

Abstract

This formal development defines μ Java, a small fragment of the programming language Java (with essentially just classes), together with a corresponding virtual machine, a specification of its bytecode verifier and a lightweight bytecode verifier. It is shown that μ Java and the given specification of the bytecode verifier are type-safe, and that the lightweight bytecode verifier is functionally equivalent to the bytecode verifier specification. See also the homepage of project Bali at <http://isabelle.in.tum.de/Bali/>.

Contents

1	JBasis	2
2	Type	3
3	Decl	4
4	TypeRel	6
5	Value	8
6	State	10
7	Term	12
8	WellForm	13
9	WellType	14
10	Eval	18
11	Conform	21
12	JTypeSafe	23
13	Example	24
14	Store of the JVM	27
15	State of the JVM	28
16	Instructions of the JVM	29
17	JVM Instruction Semantics	30
18	Program Execution in the JVM	33
19	Lifted Type Relations	34
20	Effect of instructions on the state type	44
21	The Bytecode Verifier	50
22	BV Type Safety Invariant	52
23	BV Type Safety Proof	59
24	The Lightweight Bytecode Verifier	72
25	Correctness of the LBV	76
26	Monotonicity of step and app	83
27	Completeness of the LBV	93
28	Theorem Digest	102

1 JBasis

```
(* Title:      HOL/MicroJava/J/JBasis.thy
   ID:         $Id: JBasis.thy,v 1.3 2000/09/21 08:42:58 kleing Exp $
   Author:     David von Oheimb
   Copyright   1999 TU Muenchen
```

Some auxiliary definitions.

**)*

JBasis = Main +

constdefs

```
    unique  :: "('a × 'b) list => bool"
    "unique == nodups o map fst"
```

end

2 Type

```

(* Title:      HOL/MicroJava/J/Type.thy
   ID:         $Id: Type.thy,v 1.5 2000/09/25 10:08:50 kleing Exp $
   Author:     David von Oheimb
   Copyright   1999 Technische Universitaet Muenchen

Java types
*)

Type = JBasis +

types cname (* class name *)
      vnam  (* variable or field name *)
      mname (* method name *)

arities cname, vnam, mname :: term

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

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

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

datatype ty (* any type, cf. 4.1 *)
  = 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

```

3 Decl

```
(* Title:      HOL/MicroJava/J/Decl.thy
   ID:          $Id: Decl.thy,v 1.3 2000/09/21 08:42:56 kleing Exp $
   Author:      David von Oheimb
   Copyright    1997 Technische Universitaet Muenchen
```

Class declarations and programs

**)*

Decl = Type +

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

```
types sig            (* signature of a method, cf. 8.4.2 *)
     = "mname × ty list"
```

```
'c mdecl            (* method declaration in a class *)
     = "sig × ty × 'c"
```

```
types 'c class       (* class *)
     = "cname option × fdecl list × 'c mdecl list"
     (* superclass, fields, methods*)
```

```
'c cdecl            (* class declaration, cf. 8.1 *)
     = "cname × 'c class"
```

consts

```
Object  :: cname      (* name of root class *)
ObjectC :: 'c cdecl    (* decl of root class *)
```

defs

```
ObjectC_def "ObjectC == (Object, (None, [], []))"
```

```
types 'c prog = "'c cdecl list"
```

consts

```
class      :: "'c prog => (cname ~> 'c class)"
```

```
is_class   :: "'c prog => cname => bool"
```

```
is_type    :: "'c prog => ty    => bool"
```

defs

```
class_def  "class      == map_of"
```

```
is_class_def  "is_class G C == class G C ≠ None"

primrec
  "is_type G (PrimT pt) = True"
  "is_type G (RefT t) = (case t of NullT => True | ClassT C => is_class G C)"

end
```

4 TypeRel

```

(* Title:      HOL/MicroJava/J/TypeRel.thy
   ID:         $Id: TypeRel.thy,v 1.10 2000/10/03 16:44:19 wenzelm Exp $
   Author:     David von Oheimb
   Copyright   1999 Technische Universitaet Muenchen

The relations between Java types
*)

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
  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 (HTML)
  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"

defs

(* direct subclass, cf. 8.1.3 *)
subcls1_def "subcls1 G == {(C,D). ∃ c. class G C = Some c ∧ fst c = Some D}"

consts

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

constdefs   (* auxiliary relations for recursive definitions below *)

subcls1_rel :: "(( 'c prog × cname) × ( 'c prog × cname)) set"
"subcls1_rel == {(G,C),(G',C')}. G = G' ∧ wf ((subcls1 G)^-1) ∧ G ⊢ C' <C1 C"

(* methods of a class, with inheritance, overriding and hiding, cf. 8.4.6 *)

```

```

recdef method "subcls1_rel"
  "method (G,C) = (if wf((subcls1 G)^-1) then (case class G C of None => empty
    | Some (sc,fs,ms) => (case sc of None => empty | Some D =>
      if is_class G D then method (G,D)
        else arbitrary) ++
      map_of (map (λ(s, m ).
        (s,(C,m))) ms))
    else arbitrary)"

(* list of fields of a class, including inherited and hidden ones *)
recdef fields "subcls1_rel"
  "fields (G,C) = (if wf((subcls1 G)^-1) then (case class G C of None => arbitrary
    | Some (sc,fs,ms) => map (λ(fn,ft). ((fn,C),ft)) fs@
      (case sc of None => [] | Some D =>
        if is_class G D then fields (G,D)
          else arbitrary))
    else arbitrary)"

defs

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

inductive "widen G" intrs (*widening, viz. method invocation conversion, cf. 5.3
  i.e. sort of syntactic subtyping *)
  refl          "G⊢      T ⊆ T"          (* identity conv., cf. 5.1.1 *)
  subcls "G⊢C⊆C D ==> G⊢Class C ⊆ Class D"
  null          "G⊢      NT ⊆ RefT R"

inductive "cast G" intrs (* casting conversion, cf. 5.5 / 5.1.5 *)
  (* left out casts on primitive types *)
  widen "G⊢C⊆C D ==> G⊢C ⊆? D"
  subcls "G⊢D⊆C C ==> G⊢C ⊆? D"

end

```

5 Value

```

(* Title:      HOL/MicroJava/J/Term.thy
   ID:         $Id: Value.thy,v 1.3 2000/09/22 14:28:56 kleing Exp $
   Author:     David von Oheimb
   Copyright   1999 Technische Universitaet Muenchen

Java values
*)

Value = Type +

types    loc      (* locations, i.e. abstract references on objects *)
arities  loc :: term

datatype val_      (* name non 'val' because of clash with ML token *)
  = 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 *)

types    val = val_
translations "val" <= (type) "val_"

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

6 State

```

(* Title:      HOL/MicroJava/J/State.thy
   ID:         $Id: State.thy,v 1.7 2000/09/22 14:28:54 kleing Exp $
   Author:     David von Oheimb
   Copyright   1999 Technische Universitaet Muenchen

State for evaluation of Java expressions and statements
*)

State = TypeRel + Value +

types  fields_
      = "(vname  $\times$  cname  $\rightsquigarrow$  val)" (* field name, defining class, value *)

      obj = "cname  $\times$  fields_" (* class instance with class name and fields *)

constdefs
  obj_ty      :: "obj  $\Rightarrow$  ty"
  "obj_ty obj == Class (fst obj)"

  init_vars   :: "('a  $\times$  ty) list  $\Rightarrow$  ('a  $\rightsquigarrow$  val)"
  "init_vars   == map_of o map ( $\lambda$ (n,T). (n,default_val T))"

datatype xcpt          (* exceptions *)
  = NullPointer
  | ClassCast
  | OutOfMemory

types  aheap = "loc  $\rightsquigarrow$  obj" (* "heap" used in a translation below *)
      locals = "vname  $\rightsquigarrow$  val"

      state      (* simple state, i.e. variable contents *)
      = "aheap  $\times$  locals"
      (* heap, local parameter including This *)

      xstate     (* state including exception information *)
      = "xcpt option  $\times$  state"

syntax
  heap          :: "state  $\Rightarrow$  aheap"
  locals        :: "state  $\Rightarrow$  locals"
  Norm          :: "state  $\Rightarrow$  xstate"

translations
  "heap"       $\Rightarrow$  "fst"
  "locals"     $\Rightarrow$  "snd"
  "Norm s"    == "(None,s)"

constdefs
  new_Addr      :: "aheap  $\Rightarrow$  loc  $\times$  xcpt option"

```

```

"new_Addr h == SOME (a,x). (h a = None  $\wedge$  x = None) | x = Some OutOfMemory"

raise_if      :: "bool => xcpt => xcpt option => xcpt option"
"raise_if c x xo == if c  $\wedge$  (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' ==  $\lambda$ (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  $\vee$  G $\vdash$ obj_ty (the (h (the_Addr v)))  $\preceq$  Class C"

end

```

7 Term

```

(* Title:      HOL/MicroJava/J/Term.thy
   ID:         $Id: Term.thy,v 1.8 2000/10/02 10:35:48 nipkow Exp $
   Author:     David von Oheimb
   Copyright   1999 Technische Universitaet Muenchen

Java expressions and statements
*)

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           (* local assign *) ("_::=" [ 90,90]90)
  | FAcc cname expr vname     (* field access *) ("_{_}..." [10,90,99 ]90)
  | FAss cname expr vname
    expr                     (* field ass. *) ("_{_}...::=" [10,90,99,90]90)
  | Call expr mname
    (ty list) (expr list)    (* method call*) ("..."'({_}_'" [90,99,10,10] 90)

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

```

8 WellForm

```

(* Title:      HOL/MicroJava/J/WellForm.thy
   ID:         $Id: WellForm.thy,v 1.7 2000/09/25 10:08:52 kleing Exp $
   Author:     David von Oheimb
   Copyright   1999 Technische Universitaet Muenchen

Well-formedness of Java programs
for static checks on expressions and statements, see WellType.thy

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
*)

WellForm = TypeRel +

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

constdefs
  wf_fdecl :: "'c prog => fdecl => bool"
  "wf_fdecl G ==  $\lambda$ (fn,ft). is_type G ft"

  wf_mhead :: "'c prog => sig => ty => bool"
  "wf_mhead G ==  $\lambda$ (mn,pTs) rT. ( $\forall T \in \text{set } pTs. \text{is\_type } G T$ )  $\wedge$  is_type G rT"

  wf_mdecl :: "'c wf_mb => 'c wf_mb"
  "wf_mdecl wf_mb G C ==  $\lambda$ (sig,rT,mb). wf_mhead G sig rT  $\wedge$  wf_mb G C (sig,rT,mb)"

  wf_cdecl :: "'c wf_mb => 'c prog => 'c cdecl => bool"
  "wf_cdecl wf_mb G ==
     $\lambda$ (C,(sc,fs,ms)).
    ( $\forall f \in \text{set } fs. \text{wf\_fdecl } G \ f$ )  $\wedge$  unique fs  $\wedge$ 
    ( $\forall m \in \text{set } ms. \text{wf\_mdecl } wf\_mb \ G \ C \ m$ )  $\wedge$  unique ms  $\wedge$ 
    (case sc of None => C = Object
     | Some D =>
       is_class G D  $\wedge$   $\neg \ G \vdash D \preceq C \ C \wedge$ 
       ( $\forall (sig,rT,b) \in \text{set } ms. \forall D' \ rT' \ b'.
         \text{method}(G,D) \ \text{sig} = \text{Some}(D',rT',b') \ \text{-->} \ G \vdash rT \preceq rT'$ ))"

  wf_prog :: "'c wf_mb => 'c prog => bool"
  "wf_prog wf_mb G ==
    let cs = set G in ObjectC  $\in$  cs  $\wedge$  ( $\forall c \in cs. \text{wf\_cdecl } wf\_mb \ G \ c$ )  $\wedge$  unique G"

end

```

9 WellType

```
(* Title:      HOL/MicroJava/J/WellType.thy
   ID:         $Id: WellType.thy,v 1.12 2000/09/22 14:28:56 kleing Exp $
   Author:     David von Oheimb
   Copyright   1999 Technische Universitaet Muenchen
```

Well-typedness of Java programs

the formulation of well-typedness of method calls given below (as well as the Java Specification 1.0) is a little too restrictive: It 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)

*)

WellType = Term + WellForm +

```
types   lenv (* local variables, including method parameters and This *)
        = "vname  $\rightsquigarrow$  ty"
        'c env
        = "'c prog  $\times$  lenv"
```

syntax

```
prg      :: "'c env => 'c prog"
localT   :: "'c env => (vname  $\rightsquigarrow$  ty)"
```

translations

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

consts

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

defs

```
more_spec_def "more_spec G ==  $\lambda((d,h),pTs).$   $\lambda((d',h'),pTs').$   $G \vdash d \preceq d' \wedge$ 
              list_all2 ( $\lambda T T'. G \vdash T \preceq T'$ ) pTs pTs'"
```

(* applicable methods, cf. 15.11.2.1 *)

```
appl_methds_def "appl_methds G C ==  $\lambda(mn, pTs).$ 
              {((Class md,rT),pTs') | md rT mb pTs'.
               method (G,C) (mn, pTs') = Some (md,rT,mb)  $\wedge$ 
               list_all2 ( $\lambda T T'. G \vdash T \preceq T'$ ) pTs pTs'}
```

(* maximally specific methods, cf. 15.11.2.2 *)

```
max_spec_def "max_spec G C sig == {m. m  $\in$  appl_methds G C sig  $\wedge$ 
```

```

( $\forall m' \in \text{appl\_methds } G \ C \ \text{sig.}$ 
  more_spec  $G \ m' \ m \ \rightarrow m' = m$ })"

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"

types
  java_mb = "vname list  $\times$  (vname  $\times$  ty) list  $\times$  stmt  $\times$  expr"
  (* method body with parameter names, local variables, block, result expression *)

consts
  ty_expr  :: "java_mb env => (expr       $\times$  ty      ) set"
  ty_exprs:: "java_mb env => (expr list  $\times$  ty list) set"
  wt_stmt  :: "java_mb env => stmt          set"

syntax
  ty_expr  :: "java_mb env => [expr      , ty      ] => bool" ("_  $\vdash$  _ :: _" [51,51,51]50)
  ty_exprs:: "java_mb env => [expr list, ty list] => bool" ("_  $\vdash$  _ [::] _" [51,51,51]50)
  wt_stmt  :: "java_mb env => stmt          => bool" ("_  $\vdash$  _  $\checkmark$ " [51,51 ]50)

syntax (HTML)
  ty_expr  :: "java_mb env => [expr      , ty      ] => bool" ("_  $\vdash$  _ :: _" [51,51,51]50)
  ty_exprs:: "java_mb env => [expr list, ty list] => bool" ("_  $\vdash$  _ [::] _" [51,51,51]50)
  wt_stmt  :: "java_mb env => stmt          => bool" ("_  $\vdash$  _ [ok]" [51,51 ]50)

translations
  "E  $\vdash$  e :: T" == "(e,T)  $\in$  ty_expr E"
  "E  $\vdash$  e [::] T" == "(e,T)  $\in$  ty_exprs E"
  "E  $\vdash$  c  $\checkmark$ " == "c  $\in$  wt_stmt E"

inductive "ty_expr E" "ty_exprs E" "wt_stmt E" intrs

(* well-typed expressions *)

(* cf. 15.8 *)
NewC "[| is_class (prg E) C |] ==>
  E  $\vdash$  NewC C :: Class C"

(* cf. 15.15 *)
Cast "[| E  $\vdash$  e :: Class C;
  prg E  $\vdash$  C  $\leq$ ? D |] ==>
  E  $\vdash$  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     "[| 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⊢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 ∧
  nodups pns ∧
  unique 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))"

wf_java_prog :: java_mb prog => bool
"wf_java_prog G == wf_prog wf_java_mdecl G"

end

```

10 Eval

```
(* Title:      HOL/MicroJava/J/Eval.thy
   ID:          $Id: Eval.thy,v 1.12 2000/09/22 14:28:54 kleing Exp $
   Author:      David von Oheimb
   Copyright    1999 Technische Universitaet Muenchen
```

Operational evaluation (big-step) semantics of the
execution of Java expressions and statements
)

Eval = State + WellType +

consts

```
eval  :: "java_mb prog => (xstate × expr      × val      × xstate) set"
evals :: "java_mb prog => (xstate × expr list × val list × xstate) set"
exec  :: "java_mb prog => (xstate × stmt      × xstate) set"
```

syntax

```
eval :: "[java_mb prog,xstate,expr,val,xstate] => bool "
      ("_ ⊢ _ -> _" [51,82,82,82,82] 81)
evals:: "[java_mb prog,xstate,expr list,
          val list,xstate] => bool "
      ("_ ⊢ _ -[>]_ -> _" [51,82,51,51,82] 81)
exec  :: "[java_mb prog,xstate,stmt,    xstate] => bool "
      ("_ ⊢ _ -> _" [51,82,82,82] 81)
```

syntax (HTML)

```
eval :: "[java_mb prog,xstate,expr,val,xstate] => bool "
      ("_ |- _ -> _" [51,82,82,82,82] 81)
evals:: "[java_mb prog,xstate,expr list,
          val list,xstate] => bool "
      ("_ |- _ -[>]_ -> _" [51,82,51,51,82] 81)
exec  :: "[java_mb prog,xstate,stmt,    xstate] => bool "
      ("_ |- _ -> _" [51,82,82,82] 81)
```

translations

```
"G⊢s -e > v-> (x,s')" <= "(s, e, v, x, s') ∈ eval G"
"G⊢s -e > s' " == "(s, e, v,    s') ∈ eval G"
"G⊢s -e[>]v-> (x,s')" <= "(s, e, v, x, s') ∈ evals G"
"G⊢s -e[>]v-> s' " == "(s, e, v,    s') ∈ evals G"
"G⊢s -c    -> (x,s')" <= "(s, c, x, s') ∈ exec G"
"G⊢s -c    -> s' " == "(s, c,    s') ∈ exec G"
```

inductive "eval G" "evals G" "exec G" intrs

(* evaluation of expressions *)

(* cf. 15.5 *)

XcptE "G⊢(Some xc,s) -e>arbitrary-> (Some xc,s)"

```

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

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

(* cf. 15.7.1 *)
Lit "G ⊢ Norm s -Lit v > v -> Norm s"

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

(* cf. 15.13.1, 15.2 *)
LAcc "G ⊢ Norm s -LAcc v > the (locals s v) -> Norm s"

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

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

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

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

```

(* evaluation of expression lists *)

(* cf. 15.5 *)

```

XcptEs "G⊢(Some xc,s) -e[>]arbitrary-> (Some xc,s)"

(* cf. 15.11.??? *)
Nil "G⊢Norm s0 -[] [>] []-> Norm s0"

(* cf. 15.6.4 *)
Cons "[| G⊢Norm s0 -e > v -> s1;
      G⊢ s1 -es[>]vs-> s2 |] ==>
      G⊢Norm s0 -e#es[>]v#vs-> s2"

(* execution of statements *)

(* cf. 14.1 *)
XcptS "G⊢(Some xc,s) -s0-> (Some xc,s)"

(* cf. 14.5 *)
Skip "G⊢Norm s -Skip-> Norm s"

(* cf. 14.7 *)
Expr "[| G⊢Norm s0 -e>v-> s1 |] ==>
      G⊢Norm s0 -Expr e-> s1"

(* cf. 14.2 *)
Comp "[| G⊢Norm s0 -s -> s1;
      G⊢ s1 -t -> s2|] ==>
      G⊢Norm s0 -(s;; t)-> s2"

(* cf. 14.8.2 *)
Cond "[| G⊢Norm s0 -e >v-> s1;
      G⊢ s1 -(if the_Bool v then s else t)-> s2|] ==>
      G⊢Norm s0 -(If(e) s Else t)-> s2"

(* cf. 14.10, 14.10.1 *)
Loop "[| G⊢Norm s0 -(If(e) (s;; While(e) s) Else Skip)-> s1 |] ==>
      G⊢Norm s0 -(While(e) s)-> s1"

end

```

11 Conform

```

(* Title:      HOL/MicroJava/J/Conform.thy
   ID:         $Id: Conform.thy,v 1.6 2000/09/22 14:28:53 kleing Exp $
   Author:     David von Oheimb
   Copyright   1999 Technische Universitaet Muenchen

Conformity relations for type safety of Java
*)

Conform = State + WellType +

types 'c env_ = "'c prog  $\times$  (vname  $\rightsquigarrow$  ty)" (* same as env of WellType.thy *)

constdefs

  hext :: "aheap  $\Rightarrow$  aheap  $\Rightarrow$  bool" ("_  $\leq$  | _" [51,51] 50)
  "h $\leq$  | h' ==  $\forall a C fs. h a = \text{Some}(C, fs) \rightarrow (\exists fs'. h' a = \text{Some}(C, fs'))"$ 

  conf :: "'c prog  $\Rightarrow$  aheap  $\Rightarrow$  val  $\Rightarrow$  ty  $\Rightarrow$  bool"          ("_,_  $\vdash$  _  $:: \preceq$  _" [51,51,51,51] 50)
  "G, h  $\vdash v :: \preceq T == \exists T'. \text{typeof}(\text{option\_map obj\_ty } o h) v = \text{Some } T' \wedge G \vdash T' \preceq T"$ 

  lconf :: "'c prog  $\Rightarrow$  aheap  $\Rightarrow$  ('a  $\rightsquigarrow$  val)  $\Rightarrow$  ('a  $\rightsquigarrow$  ty)  $\Rightarrow$  bool"
                                     ("_,_  $\vdash$  _  $:: \preceq$ ] _" [51,51,51,51] 50)
  "G, h  $\vdash vs :: \preceq$ ] Ts ==  $\forall n T. Ts n = \text{Some } T \rightarrow (\exists v. vs n = \text{Some } v \wedge G, h \vdash v :: \preceq T)"$ 

  oconf :: "'c prog  $\Rightarrow$  aheap  $\Rightarrow$  obj  $\Rightarrow$  bool" ("_,_  $\vdash$  _  $\surd$ " [51,51,51] 50)
  "G, h  $\vdash obj \surd == G, h \vdash \text{snd obj} :: \preceq$ ]map_of (fields (G, fst obj))"

  hconf :: "'c prog  $\Rightarrow$  aheap  $\Rightarrow$  bool" ("_  $\vdash h$  _  $\surd$ " [51,51] 50)
  "G  $\vdash h h \surd == \forall a obj. h a = \text{Some } obj \rightarrow G, h \vdash obj \surd"$ 

  conforms :: "state  $\Rightarrow$  java_mb env_  $\Rightarrow$  bool" ("_  $:: \preceq$  _" [51,51] 50)
  "s  $:: \preceq E == \text{prg } E \vdash h \text{ heap } s \surd \wedge \text{prg } E, \text{heap } s \vdash \text{locals } s :: \preceq$ ]localT E"

syntax (HTML)
  hext      :: "aheap  $\Rightarrow$  aheap  $\Rightarrow$  bool"
              ("_  $\leq$  | _" [51,51] 50)

  conf      :: "'c prog  $\Rightarrow$  aheap  $\Rightarrow$  val  $\Rightarrow$  ty  $\Rightarrow$  bool"
              ("_,_ | - _  $:: \leq$  _" [51,51,51,51] 50)

  lconf     :: "'c prog  $\Rightarrow$  aheap  $\Rightarrow$  ('a  $\rightsquigarrow$  val)  $\Rightarrow$  ('a  $\rightsquigarrow$  ty)  $\Rightarrow$  bool"
              ("_,_ | - _  $:: \leq$ ] _" [51,51,51,51] 50)

  oconf     :: "'c prog  $\Rightarrow$  aheap  $\Rightarrow$  obj  $\Rightarrow$  bool"
              ("_,_ | - _ [ok]" [51,51,51] 50)

  hconf     :: "'c prog  $\Rightarrow$  aheap  $\Rightarrow$  bool"
              ("_ | - h _ [ok]" [51,51] 50)

```

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

12 *JTypeSafe*

```
(* Title:      HOL/MicroJava/J/JTypeSafe.thy
   ID:         $Id: JTypeSafe.thy,v 1.1 1999/11/11 11:24:03 nipkow Exp $
   Author:     David von Oheimb
   Copyright   1999 Technische Universitaet Muenchen
```

Type Safety of Java

*)

JTypeSafe = Eval + Conform

13 Example

```
(* Title:      isabelle/Bali/Example.thy
   ID:         $Id: Example.thy,v 1.5 2000/09/21 08:42:57 kleing Exp $
   Author:     David von Oheimb
   Copyright   1997 Technische Universitaet Muenchen
```

The following example Bali program includes:

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

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

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

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

Example = Eval +

```
datatype cnam_ = Base_ | Ext_
datatype vnam_ = vee_ | x_ | e_
```

consts

```
cnam_ :: "cnam_ => cname"
vnam_ :: "vnam_ => vnam"
```

rules (* cnam_ and vnam_ are intended to be isomorphic to cnam and vnam *)

```
inj_cnam_ "(cnam_ x = cnam_ y) = (x = y)"
inj_vnam_ "(vnam_ x = vnam_ y) = (x = y)"
```

```
surj_cnam_ "∃ m. n = cnam_ m"
surj_vnam_ "∃ m. n = vnam_ m"
```

syntax

```

Base, Ext    :: cname
vee, x, e    :: vname

```

translations

```

"Base" == "cnam_ Base_"
"Ext"  == "cnam_ Ext_"
"vee"  == "VName (vnam_ vee_)"
"x"    == "VName (vnam_ x_)"
"e"    == "VName (vnam_ e_)"

```

rules

```

Base_not_Object "Base ≠ Object"
Ext_not_Object  "Ext  ≠ Object"

```

consts

```

foo_Base      :: java_mb
foo_Ext       :: java_mb
BaseC, ExtC   :: java_mb cdecl
test          :: stmt
foo           :: mname
a,b           :: loc

```

defs

```

foo_Base_def "foo_Base == ([x],[],Skip,LAcc x)"
BaseC_def   "BaseC == (Base, (Some Object,
                               [(vee, PrimT Boolean)],
                               [((foo,[Class Base]),Class Base,foo_Base))])"
foo_Ext_def "foo_Ext == ([x],[],Expr( {Ext}Cast Ext
                                       (LAcc x)..vee:=Lit (Intg #1)),
                          Lit Null)"
ExtC_def    "ExtC == (Ext, (Some Base ,
                           [(vee, PrimT Integer)],
                           [((foo,[Class Base]),Class Ext,foo_Ext))])"

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

```

syntax

```

NP          :: xcpt
tprg        :: java_mb prog
obj1, obj2  :: obj
s0,s1,s2,s3,s4:: state

```

translations

```

"NP" == "NullPointer"
"tprg" == "[ObjectC, BaseC, ExtC]"

```

```

"obj1"    <= "(Ext, empty((vee, Base)⊢Bool False)
              ((vee, Ext )⊢Intg #0))"
"s0" == " Norm      (empty, empty)"
"s1" == " Norm      (empty(a⊢obj1),empty(e⊢Addr a))"
"s2" == " Norm      (empty(a⊢obj1),empty(x⊢Null)(This⊢Addr a))"
"s3" == "(Some NP, empty(a⊢obj1),empty(e⊢Addr a))"
end

```

14 Store of the JVM

theory *Store* = *Conform*:

The JVM builds on many notions already defined in Java. Conform provides notions for the type safety proof of the Bytecode Verifier.

constdefs

```
newref :: "('a  $\rightsquigarrow$  'b) => 'a"
"newref s == SOME v. s v = None"
```

lemma *newref_None*:

```
"hp x = None ==> hp (newref hp) = None"
by (auto intro: someI2_ex simp add: newref_def)
```

end

15 State of the JVM

theory JVMState = Store:

frame stack :

types

```

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

```

```

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

```

exceptions:

constdefs

```

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

```

runtime state:

types

```

jvm_state = "xcpt option × aheap × frame list"

```

dynamic method lookup:

constdefs

```

dyn_class      :: "'code prog × sig × cname => cname"
"dyn_class == λ(G,sig,C). fst(the(method(G,C) sig))"

```

end

16 Instructions of the JVM

theory *JVMInstructions* = *JVMState*:

datatype

```

  instr = Load nat
        | Store nat
        | Bipush int
        | Aconst_null
        | New cname
        | Getfield vname cname
        | Putfield vname cname
        | Checkcast cname
        | Invoke cname mname "(ty list)"
        | Return
        | Pop
        | Dup
        | Dup_x1
        | Dup_x2
        | Swap
        | IAdd
        | Goto int
        | Ifcmpeq int

```

types

```

  bytecode = "instr list"
  jvm_prog = "(nat × bytecode) prog"

```

end

17 JVM Instruction Semantics

theory JVMExecInstr = JVMInstructions + JVMState:

consts

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

primrec

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

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

```
"exec_instr (Bipush ival) G hp stk vars Cl sig pc frs =
  (None, hp, (Intg ival # stk, vars, Cl, sig, pc+1)#frs)"
```

```
"exec_instr Aconst_null G hp stk vars Cl sig pc frs =
  (None, hp, (Null # stk, vars, Cl, sig, pc+1)#frs)"
```

```
"exec_instr (New C) G hp stk vars Cl sig pc frs =
  (let xp'      = raise_xcpt (∀x. hp x ≠ None) OutOfMemory;
    oref        = newref hp;
    fs          = init_vars (fields(G,C));
    hp'         = if xp'=None then hp(oref ↦ (C,fs)) else hp;
    stk'        = if xp'=None then (Addr oref)#stk else stk
  in
  (xp', hp', (stk', vars, Cl, sig, pc+1)#frs))"

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

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

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

```

(xp', hp, (stk', vars, Cl, sig, pc+1)#frs))"

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

"exec_instr Return G hp stk0 vars Cl sig0 pc frs =
  (if frs=[] then
    (None, hp, [])
  else
    let val = hd stk0; (stk,loc,C,sig,pc) = hd frs
    in
      (None, hp, (val#stk,loc,C,sig,pc)#tl frs))"

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

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

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

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

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

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

"exec_instr Ifcmpeq i) G hp stk vars Cl sig pc frs =
  (let (val1,val2) = (hd stk, hd (tl stk));
  pc' = if val1 = val2 then nat(int pc+i) else pc+1

```

```

      in
        (None, hp, (tl (tl stk), vars, Cl, sig, pc')#frs))"

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

end

```

18 Program Execution in the JVM

```

theory JVMExec = JVMExecInstr :

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

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

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

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

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

syntax (HTML)
  exec_all :: "[jvm_prog, jvm_state, jvm_state] => bool"
    ("_ |- _ -jvm-> _" [61, 61, 61] 60)

end

```

19 Lifted Type Relations

theory *Convert* = *JVMExec*:

The supertype relation lifted to type err, type lists and state types.

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

types

```
locvars_type = "ty err list"
opstack_type = "ty list"
state_type   = "opstack_type × locvars_type"
```

consts

```
strict :: "('a => 'b err) => ('a err => 'b err)"
```

primrec

```
"strict f Err      = Err"
"strict f (Ok x) = f x"
```

consts

```
val :: "'a err => 'a"
```

primrec

```
"val (Ok s) = s"
```

constdefs

```
lift_top :: "('a => 'b => bool) => ('a err => 'b err => bool)"
"lift_top P a' a == case a of
  Err   => True
| Ok t => (case a' of Err => False | Ok t' => P t' t)"
```

```
lift_bottom :: "('a => 'b => bool) => ('a option => 'b option => bool)"
"lift_bottom P a' a ==
  case a' of
    None   => True
  | Some t' => (case a of None => False | Some t => P t' t)"
```

```
sup_ty_opt :: "[code prog, ty err, ty err] => bool"
(" _ ⊢ _ ≤o _" [71,71] 70)
"sup_ty_opt G == lift_top (λt t'. G ⊢ t ≤ t')"
```

```
sup_loc :: "[code prog, locvars_type, locvars_type] => bool"
(" _ ⊢ _ ≤l _" [71,71] 70)
"G ⊢ LT ≤l LT' == list_all2 (λt t'. (G ⊢ t ≤o t')) LT LT'"
```

```
sup_state :: "[code prog, state_type, state_type] => bool"
(" _ ⊢ _ ≤s _" [71,71] 70)
"G ⊢ s ≤s s' ==
  (G ⊢ map Ok (fst s) ≤l map Ok (fst s')) ∧ G ⊢ snd s ≤l snd s'"
```

```

sup_state_opt :: "[code prog, state_type option, state_type option] => bool"
               ("_ ⊢ _ <=' _" [71,71] 70)
"sup_state_opt G == lift_bottom (λt t'. G ⊢ t <=s t')"
```

syntax (HTML)

```

sup_ty_opt    :: "[code prog, ty err, ty err] => bool"
               ("_ |- _ <=o _")
sup_loc       :: "[code prog, locvars_type, locvars_type] => bool"
               ("_ |- _ <=l _" [71,71] 70)
sup_state     :: "[code prog, state_type, state_type] => bool"
               ("_ |- _ <=s _" [71,71] 70)
sup_state_opt :: "[code prog, state_type option, state_type option] => bool"
               ("_ |- _ <=' _" [71,71] 70)
```

lemma not_Err_eq [iff]:

```

"(x ≠ Err) = (∃a. x = Ok a)"
by (cases x) auto
```

lemma not_Some_eq [iff]:

```

"(∀y. x ≠ Ok y) = (x = Err)"
by (cases x) auto
```

lemma lift_top_refl [simp]:

```

"[| !!x. P x x |] ==> lift_top P x x"
by (simp add: lift_top_def split: err.splits)
```

lemma lift_top_trans [trans]:

```

"[| !!x y z. [| P x y; P y z |] ==> P x z; lift_top P x y; lift_top P y z |]
==> lift_top P x z"
```

proof -

```

  assume [trans]: "!!x y z. [| P x y; P y z |] ==> P x z"
  assume a: "lift_top P x y"
  assume b: "lift_top P y z"

  { assume "z = Err"
    hence ?thesis by (simp add: lift_top_def)
  } note z_none = this

  { assume "x = Err"
    with a b
    have ?thesis
      by (simp add: lift_top_def split: err.splits)
  } note x_none = this

  { fix r t
    assume x: "x = Ok r" and z: "z = Ok t"
    with a b
    obtain s where y: "y = Ok s"
      by (simp add: lift_top_def split: err.splits)
  }
```

```

    from a x y
    have "P r s" by (simp add: lift_top_def)
    also
    from b y z
    have "P s t" by (simp add: lift_top_def)
    finally
    have "P r t" .

    with x z
    have ?thesis by (simp add: lift_top_def)
  }

  with x_none z_none
  show ?thesis by blast
qed

lemma lift_top_Err_any [simp]:
  "lift_top P Err any = (any = Err)"
  by (simp add: lift_top_def split: err.splits)

lemma lift_top_Ok_Ok [simp]:
  "lift_top P (Ok a) (Ok b) = P a b"
  by (simp add: lift_top_def split: err.splits)

lemma lift_top_any_Ok [simp]:
  "lift_top P any (Ok b) = ( $\exists$ a. any = Ok a  $\wedge$  P a b)"
  by (simp add: lift_top_def split: err.splits)

lemma lift_top_Ok_any:
  "lift_top P (Ok a) any = (any = Err  $\vee$  ( $\exists$ b. any = Ok b  $\wedge$  P a b))"
  by (simp add: lift_top_def split: err.splits)

lemma lift_bottom_refl [simp]:
  "[| !!x. P x x |] ==> lift_bottom P x x"
  by (simp add: lift_bottom_def split: option.splits)

lemma lift_bottom_trans [trans]:
  "[| !!x y z. [| P x y; P y z |] ==> P x z;
    lift_bottom P x y; lift_bottom P y z |]
  ==> lift_bottom P x z"
proof -
  assume [trans]: "!!x y z. [| P x y; P y z |] ==> P x z"
  assume a: "lift_bottom P x y"
  assume b: "lift_bottom P y z"

  { assume "x = None"
    hence ?thesis by (simp add: lift_bottom_def)
  } note z_none = this

  { assume "z = None"
    with a b

```

```

    have ?thesis
      by (simp add: lift_bottom_def split: option.splits)
  } note x_none = this

{ fix r t
  assume x: "x = Some r" and z: "z = Some t"
  with a b
  obtain s where y: "y = Some s"
    by (simp add: lift_bottom_def split: option.splits)

  from a x y
  have "P r s" by (simp add: lift_bottom_def)
  also
  from b y z
  have "P s t" by (simp add: lift_bottom_def)
  finally
  have "P r t" .

  with x z
  have ?thesis by (simp add: lift_bottom_def)
}

with x_none z_none
show ?thesis by blast
qed

lemma lift_bottom_any_None [simp]:
  "lift_bottom P any None = (any = None)"
  by (simp add: lift_bottom_def split: option.splits)

lemma lift_bottom_Some_Some [simp]:
  "lift_bottom P (Some a) (Some b) = P a b"
  by (simp add: lift_bottom_def split: option.splits)

lemma lift_bottom_any_Some [simp]:
  "lift_bottom P (Some a) any = ( $\exists b. \text{any} = \text{Some } b \wedge P a b$ )"
  by (simp add: lift_bottom_def split: option.splits)

lemma lift_bottom_Some_any:
  "lift_bottom P any (Some b) = ( $\text{any} = \text{None} \vee (\exists a. \text{any} = \text{Some } a \wedge P a b)$ )"
  by (simp add: lift_bottom_def split: option.splits)

theorem sup_ty_opt_refl [simp]:
  " $G \vdash t \leq_o t$ "
  by (simp add: sup_ty_opt_def)

theorem sup_loc_refl [simp]:
  " $G \vdash t \leq_l t$ "
  by (induct t, auto simp add: sup_loc_def)

```

```

theorem sup_state_refl [simp]:
  "G ⊢ s ≤s s"
  by (simp add: sup_state_def)

theorem sup_state_opt_refl [simp]:
  "G ⊢ s ≤' s"
  by (simp add: sup_state_opt_def)

theorem anyConvErr [simp]:
  "(G ⊢ Err ≤o any) = (any = Err)"
  by (simp add: sup_ty_opt_def)

theorem OkanyConvOk [simp]:
  "(G ⊢ (Ok ty') ≤o (Ok ty)) = (G ⊢ ty' ≤ ty)"
  by (simp add: sup_ty_opt_def)

theorem sup_ty_opt_Ok:
  "G ⊢ a ≤o (Ok b) ==> ∃ x. a = Ok x"
  by (clarsimp simp add: sup_ty_opt_def)

lemma widen_PrimT_conv1 [simp]:
  "[| G ⊢ S ≤ T; S = PrimT x |] ==> T = PrimT x"
  by (auto elim: widen.elims)

theorem sup_PTS_eq:
  "(G ⊢ Ok (PrimT p) ≤o X) = (X=Err ∨ X = Ok (PrimT p))"
  by (auto simp add: sup_ty_opt_def lift_top_Ok_any)

theorem sup_loc_Nil [iff]:
  "(G ⊢ [] ≤l XT) = (XT=[])"
  by (simp add: sup_loc_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 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 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::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 @ [1]) ≤=1 (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 ≤=1 rev list" "G ⊢ l ≤=o a"
      by (simp add: sup_loc_append)

    thus "G ⊢ (l # ls) ≤=1 b"
      by (simp add: Cons IH)
  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 ≤=1 y) -->
    (G ⊢ x[n := a] ≤=1 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) ≤=1 y" "n < length y"
    with IH
    show "G ⊢ (l # ls)[n := a] ≤=1 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)

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 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 sup_loc_append)

theorem sup_state_Cons1:
  "(G ⊢ (x#xt, a) <=s (yt, b)) =
   (∃y yt'. yt=y#yt' ∧ (G ⊢ x ≼ y) ∧ (G ⊢ (xt,a) <=s (yt',b)))"
  by (auto simp add: sup_state_def map_eq_Cons)

theorem sup_state_Cons2:
  "(G ⊢ (xt, a) <=s (y#yt, b)) =
   (∃x xt'. xt=x#xt' ∧ (G ⊢ x ≼ y) ∧ (G ⊢ (xt',a) <=s (yt,b)))"
  by (auto simp add: sup_state_def 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)

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)
qed

lemma sup_state_opt_None_any [iff]:
  "(G ⊢ None <=' any) = True"
  by (simp add: sup_state_opt_def lift_bottom_def)

lemma sup_state_opt_any_None [iff]:
  "(G ⊢ any <=' None) = (any = None)"
  by (simp add: sup_state_opt_def)

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

lemma sup_state_opt_any_Some [iff]:
  "(G ⊢ (Some a) <=' any) = (∃b. any = Some b ∧ G ⊢ a <=s b)"
  by (simp add: sup_state_opt_def)

lemma sup_state_opt_Some_any:
  "(G ⊢ any <=' (Some b)) = (any = None ∨ (∃a. any = Some a ∧ G ⊢ a <=s b))"
  by (simp add: sup_state_opt_def lift_bottom_Some_any)

theorem sup_ty_opt_trans [trans]:
  "[| G ⊢ a <=o b; G ⊢ b <=o c |] ==> G ⊢ a <=o c"
  by (auto intro: lift_top_trans widen_trans simp add: sup_ty_opt_def)

theorem sup_loc_trans [trans]:
  "[| G ⊢ a <=l b; G ⊢ b <=l c |] ==> G ⊢ a <=l c"

```

proof -

assume G : " $G \vdash a \leq_l b$ " " $G \vdash b \leq_l c$ "

hence " $\forall n. n < \text{length } a \rightarrow (G \vdash (a!n) \leq_o (c!n))$ "

proof (intro strip)

fix n

assume n : " $n < \text{length } a$ "

with G

have " $G \vdash (a!n) \leq_o (b!n)$ "

by - (rule sup_loc_nth)

also

from n G

have " $G \vdash \dots \leq_o (c!n)$ "

by - (rule sup_loc_nth, auto dest: sup_loc_length)

finally

show " $G \vdash (a!n) \leq_o (c!n)$ " .

qed

with G

show ?thesis

by (auto intro!: all_nth_sup_loc [rule_format] dest!: sup_loc_length)

qed

theorem sup_state_trans [trans]:

" $[G \vdash a \leq_s b; G \vdash b \leq_s c] \implies G \vdash a \leq_s c$ "

by (auto intro: sup_loc_trans simp add: sup_state_def)

theorem sup_state_opt_trans [trans]:

" $[G \vdash a \leq' b; G \vdash b \leq' c] \implies G \vdash a \leq' c$ "

by (auto intro: lift_bottom_trans sup_state_trans simp add: sup_state_opt_def)

end

20 Effect of instructions on the state type

theory Step = Convert:

Effect of instruction on the state type:

consts

```
step' :: "instr × jvm_prog × state_type => state_type"
```

```
reodef step' "{}"
```

```

"step' (Load idx, G, (ST, LT))           = (val (LT ! idx) # ST, LT)"
"step' (Store idx, G, (ts#ST, LT))       = (ST, LT[idx:= Ok ts])"
"step' (Bipush i, G, (ST, LT))           = (PrimT Integer # ST, LT)"
"step' (Aconst_null, G, (ST, LT))        = (NT#ST,LT)"
"step' (Getfield F C, G, (oT#ST, LT))    = (snd (the (field (G,C) F)) # ST, LT)"
"step' (Putfield F C, G, (vT#oT#ST, LT)) = (ST,LT)"
"step' (New C, G, (ST,LT))                = (Class C # ST, LT)"
"step' (Checkcast C, G, (RefT rt#ST,LT)) = (Class C # ST,LT)"
"step' (Pop, G, (ts#ST,LT))               = (ST,LT)"
"step' (Dup, G, (ts#ST,LT))               = (ts#ts#ST,LT)"
"step' (Dup_x1, G, (ts1#ts2#ST,LT))      = (ts1#ts2#ts1#ST,LT)"
"step' (Dup_x2, G, (ts1#ts2#ts3#ST,LT))  = (ts1#ts2#ts3#ts1#ST,LT)"
"step' (Swap, G, (ts1#ts2#ST,LT))        = (ts2#ts1#ST,LT)"
"step' (IAdd, G, (PrimT Integer#PrimT Integer#ST,LT))
                                           = (PrimT Integer#ST,LT)"
"step' (Ifcmpeq b, G, (ts1#ts2#ST,LT))   = (ST,LT)"
"step' (Goto b, G, s)                     = s"

```

```
"step' (Invoke C mn fpTs, G, (ST,LT)) = (let ST' = drop (length fpTs) ST
  in (fst (snd (the (method (G,C) (mn,fpTs))))#(tl ST'),LT))"
```

constdefs

```
step :: "instr => jvm_prog => state_type option => state_type option"
"step i G == option_map (\s. step' (i,G,s))"
```

Conditions under which step is applicable:

consts

```
app' :: "instr × jvm_prog × ty × state_type => bool"
```

```
recdef app' "{}"
```

[illegible]

```

fst (the (field (G,C) F)) = C ∧
G ⊢ oT ≤ (Class C) ∧
G ⊢ vT ≤ (snd (the (field (G,C) F)))"
"app' (New C, G, rT, s) = (is_class G C)"
"app' (Checkcast C, G, rT, (RefT rt#ST,LT)) = (is_class G C)"
"app' (Pop, G, rT, (ts#ST,LT)) = True"
"app' (Dup, G, rT, (ts#ST,LT)) = True"
"app' (Dup_x1, G, rT, (ts1#ts2#ST,LT)) = True"
"app' (Dup_x2, G, rT, (ts1#ts2#ts3#ST,LT)) = True"
"app' (Swap, G, rT, (ts1#ts2#ST,LT)) = True"
"app' (IAdd, G, rT, (PrimT Integer#PrimT Integer#ST,LT))
= True"
"app' (Ifcmpeq b, G, rT, (ts#ts'#ST,LT)) = ((∃ p. ts = PrimT p ∧ ts' = PrimT p) ∨
(∃ r r'. ts = RefT r ∧ ts' = RefT r'))"
"app' (Goto b, G, rT, s) = True"
"app' (Return, G, rT, (T#ST,LT)) = (G ⊢ T ≤ rT)"
"app' (Invoke C mn fpTs, G, rT, s) =
(length fpTs < length (fst s) ∧
(let apTs = rev (take (length fpTs) (fst s));
X = hd (drop (length fpTs) (fst s))
in
G ⊢ X ≤ Class C ∧ method (G,C) (mn,fpTs) ≠ None ∧
(∀ (aT,fT) ∈ set(zip apTs fpTs). G ⊢ aT ≤ fT)))"

"app' (i,G,rT,s) = False"

```

constdefs

```

app :: "instr => jvm_prog => ty => state_type option => bool"
"app i G rT s == case s of None => True | Some t => app' (i,G,rT,t)"

```

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 (Bipush i) pc = [pc+1]"
"succs (Aconst_null) 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          = []"
"succs (Invoke C mn fpTs) pc = [pc+1]"
```

```
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 2)" by simp
  hence * : "length a = 0 ∨ length a = 1 ∨ length a = 2"
    by (auto simp add: less_Suc_eq)

  {
    fix x
    assume "length x = 1"
    hence "∃ l. x = [l]" by - (cases x, auto)
  } note 0 = this

  have "length a = 2 ==> ∃ l l'. a = [l,l']" by (cases a, auto dest: 0)
  with * show ?thesis by (auto dest: 0)
qed
```

simp rules for app

```
lemma appNone[simp]:
  "app i G rT None = True"
  by (simp add: app_def)
```

```
lemma appLoad[simp]:
  "(app (Load idx) G rT (Some s)) = (idx < length (snd s) ∧ (snd s) ! idx ≠ Err)"
  by (simp add: app_def)
```

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

```
lemma appBipush[simp]:
  "(app (Bipush i) G rT (Some s)) = True"
  by (simp add: app_def)
```

```
lemma appAconst[simp]:
  "(app Aconst_null G rT (Some s)) = True"
  by (simp add: app_def)
```

```
lemma appGetField[simp]:
  "(app (Getfield F C) G rT (Some s)) =
    (∃ oT vT ST LT. s = (oT#ST, LT) ∧ is_class G C ∧
```

```

    field (G,C) F = Some (C,vT) ∧ G ⊢ oT ≼ (Class C))"
  by (cases s, cases "2 < length (fst s)", auto dest!: 1 2 simp add: app_def)

lemma appPutField[simp]:
  "(app (Putfield F C) G rT (Some s)) =
    (∃ vT vT' oT ST LT. s = (vT#oT#ST, LT) ∧ is_class G C ∧
      field (G,C) F = Some (C, vT') ∧ G ⊢ oT ≼ (Class C) ∧ G ⊢ vT ≼ vT')"
  by (cases s, cases "2 < length (fst s)", auto dest!: 1 2 simp add: app_def)

lemma appNew[simp]:
  "(app (New C) G rT (Some s)) = is_class G C"
  by (simp add: app_def)

lemma appCheckcast[simp]:
  "(app (Checkcast C) G rT (Some s)) = (∃ rT ST LT. s = (RefT rT#ST,LT) ∧ is_class G C)"
  by (cases s, cases "fst s", simp add: app_def)
    (cases "hd (fst s)", auto simp add: app_def)

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

lemma appDup[simp]:
  "(app Dup G rT (Some s)) = (∃ ts ST LT. s = (ts#ST,LT))"
  by (cases s, cases "2 < length (fst s)", auto dest: 1 2 simp add: app_def)

lemma appDup_x1[simp]:
  "(app Dup_x1 G rT (Some s)) = (∃ ts1 ts2 ST LT. s = (ts1#ts2#ST,LT))"
  by (cases s, cases "2 < length (fst s)", auto dest: 1 2 simp add: app_def)

lemma appDup_x2[simp]:
  "(app Dup_x2 G rT (Some s)) = (∃ ts1 ts2 ts3 ST LT. s = (ts1#ts2#ts3#ST,LT))"
  by (cases s, cases "2 < length (fst s)", auto dest: 1 2 simp add: app_def)

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

lemma appIAdd[simp]:
  "(app IAdd G rT (Some s)) = (∃ ST LT. s = (PrimT Integer#PrimT Integer#ST,LT))"
  (is "?app s = ?P s")
proof (cases (open) s)
  case Pair
  have "?app (a,b) = ?P (a,b)"
  proof (cases "a")
    fix t ts assume a: "a = t#ts"
    show ?thesis

```

```

proof (cases t)
  fix p assume p: "t = PrimT p"
  show ?thesis
  proof (cases p)
    assume ip: "p = Integer"
    show ?thesis
    proof (cases ts)
      fix t' ts' assume t': "ts = t' # ts'"
      show ?thesis
      proof (cases t')
        fix p' assume "t' = PrimT p'"
        with t' ip p a
          show ?thesis by - (cases p', auto simp add: app_def)
      qed (auto simp add: a p ip t' app_def)
      qed (auto simp add: a p ip app_def)
      qed (auto simp add: a p app_def)
      qed (auto simp add: a app_def)
    qed (auto simp add: app_def)
  with Pair show ?thesis by simp
qed

lemma appIfcmpeq[simp]:
  "app (Ifcmpeq b) G rT (Some s) = ( $\exists$  ts1 ts2 ST LT. s = (ts1#ts2#ST,LT)  $\wedge$ 
  ( $\exists$  p. ts1 = PrimT p  $\wedge$  ts2 = PrimT p)  $\vee$  ( $\exists$  r r'. ts1 = RefT r  $\wedge$  ts2 = RefT r')))"
  by (cases s, cases "2 < length (fst s)", auto dest: 1 2 simp add: app_def)

lemma appReturn[simp]:
  "app Return G rT (Some s) = ( $\exists$  T ST LT. s = (T#ST,LT)  $\wedge$  (G  $\vdash$  T  $\preceq$  rT))"
  by (cases s, cases "2 < length (fst s)", auto dest: 1 2 simp add: app_def)

lemma appGoto[simp]:
  "app (Goto branch) G rT (Some s) = True"
  by (simp add: app_def)

lemma appInvoke[simp]:
  "app (Invoke C mn fpTs) G rT (Some s) = ( $\exists$  apTs X ST LT mD' rT' b'.
  s = ((rev apTs) @ (X # ST), LT)  $\wedge$  length apTs = length fpTs  $\wedge$ 
  G  $\vdash$  X  $\preceq$  Class C  $\wedge$  ( $\forall$  (aT,fT)  $\in$  set(zip apTs fpTs). G  $\vdash$  aT  $\preceq$  fT)  $\wedge$ 
  method (G,C) (mn,fpTs) = Some (mD', rT', b'))" (is "?app s = ?P s")
proof (cases (open) s)
  case Pair
  have "?app (a,b) ==> ?P (a,b)"
  proof -
    assume app: "?app (a,b)"
    hence "a = (rev (rev (take (length fpTs) a))) @ (drop (length fpTs) a)  $\wedge$ 
      length fpTs < length a" (is "?a  $\wedge$  ?l")
    by (auto simp add: app_def)
    hence "?a  $\wedge$  0 < length (drop (length fpTs) a)" (is "?a  $\wedge$  ?l")
    by auto
    hence "?a  $\wedge$  ?l  $\wedge$  length (rev (take (length fpTs) a)) = length fpTs"

```

```

    by (auto simp add: min_def)
  hence "∃ apTs ST. a = rev apTs @ ST ∧ length apTs = length fpTs ∧ 0 < length ST"
    by blast
  hence "∃ apTs ST. a = rev apTs @ ST ∧ length apTs = length fpTs ∧ ST ≠ []"
    by blast
  hence "∃ apTs ST. a = rev apTs @ ST ∧ length apTs = length fpTs ∧
    (∃ X ST'. ST = X#ST')"
    by (simp add: neq_Nil_conv)
  hence "∃ apTs X ST. a = rev apTs @ X # ST ∧ length apTs = length fpTs"
    by blast
  with app
  show ?thesis by (auto simp add: app_def) blast
qed
with Pair have "?app s ==> ?P s" by simp
thus ?thesis by (auto simp add: app_def)
qed

```

```

lemma step_Some:
  "step i G (Some s) ≠ None"
  by (simp add: step_def)

```

```

lemma step_None [simp]:
  "step i G None = None"
  by (simp add: step_def)

```

```

end

```

21 The Bytecode Verifier

theory BVSpec = Step:

types

```
method_type = "state_type option list"
class_type   = "sig => method_type"
prog_type    = "cname => class_type"
```

constdefs

```
wt_instr :: "[instr,jvm_prog,ty,method_type,p_count,p_count] => bool"
"wt_instr i G rT phi max_pc pc ==
  app i G rT (phi!pc) ∧
  (∀ pc' ∈ set (succs i pc). pc' < max_pc ∧ (G ⊢ step i G (phi!pc) <= ' phi!pc'))"
```

```
wt_start :: "[jvm_prog,cname,ty list,nat,method_type] => bool"
```

```
"wt_start G C pTs mxl phi ==
  G ⊢ Some ([],(Ok (Class C))#((map Ok pTs))@(replicate mxl Err)) <= ' phi!0"
```

```
wt_method :: "[jvm_prog,cname,ty list,ty,nat,instr list,method_type] => bool"
```

```
"wt_method G C pTs rT mxl ins phi ==
  let max_pc = length ins
  in
  0 < max_pc ∧ wt_start G C pTs mxl phi ∧
  (∀ pc. pc < max_pc --> wt_instr (ins ! pc) G rT phi max_pc pc)"
```

```
wt_jvm_prog :: "[jvm_prog,prog_type] => bool"
```

```
"wt_jvm_prog G phi ==
  wf_prog (λG C (sig,rT,maxl,b).
    wt_method G C (snd sig) rT maxl b (phi C sig)) G"
```

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; method (G,C) sig = Some (C,rT,maxl,ins); pc < length ins |]
==> wt_instr (ins!pc) G rT (phi C sig) (length ins) pc"
```

by (unfold wt_jvm_prog_def, drule method_wf_mdecl, simp, simp add: wf_mdecl_def wt_method_def)

lemma wt_jvm_prog_impl_wt_start:

```
"[| wt_jvm_prog G phi; method (G,C) sig = Some (C,rT,maxl,ins) |] ==>
  0 < (length ins) ∧ wt_start G C (snd sig) maxl (phi C sig)"
```

by (unfold wt_jvm_prog_def, drule method_wf_mdecl, simp, simp add: wf_mdecl_def wt_method_def)

lemma

```

"succs i pc = [pc+1] ==> wt_instr i G rT phi max_pc pc =
  (app i G rT (phi!pc) ∧ pc+1 < max_pc ∧ (G ⊢ step i G (phi!pc) <= ' phi!(pc+1)))"
by (simp add: wt_instr_def)

end

```

22 BV Type Safety Invariant

theory Correct = BVSpec:

constdefs

```

approx_val :: "[jvm_prog, aheap, val, ty err] => bool"
"approx_val G h v any == case any of Err => True | Ok T => G, h ⊢ v :: ≤ T"

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

approx_stk :: "[jvm_prog, aheap, opstack, opstack_type] => bool"
"approx_stk G hp stk ST == approx_loc G hp stk (map Ok ST)"

correct_frame :: "[jvm_prog, aheap, state_type, nat, bytecode] => frame => bool"
"correct_frame G hp == λ(ST, LT) maxl ins (stk, loc, C, sig, pc).
    approx_stk G hp stk ST ∧ approx_loc G hp loc LT ∧
    pc < length ins ∧ length loc = length (snd sig) + maxl + 1"

correct_frame_opt :: "[jvm_prog, aheap, state_type option, nat, bytecode]
    => frame => bool"
"correct_frame_opt G hp s ==
    case s of None => λmaxl ins f. False | Some t => correct_frame G hp t"

```

consts

```

correct_frames :: "[jvm_prog, aheap, prog_type, ty, sig, frame list] => bool"

```

primrec

```

"correct_frames G hp phi rT0 sig0 [] = True"

"correct_frames G hp phi rT0 sig0 (f#frs) =
    (let (stk, loc, C, sig, pc) = f in
    (∃ ST LT rT maxl ins.
        phi C sig ! pc = Some (ST, LT) ∧
        method (G, C) sig = Some(C, rT, (maxl, ins)) ∧
        (∃ C' mn pTs k. pc = k+1 ∧ ins!k = (Invoke C' mn pTs) ∧
            (mn, pTs) = sig0 ∧
            (∃ apTs D ST' LT'.
                (phi C sig)!k = Some ((rev apTs) @ (Class D) # ST', LT')) ∧
                length apTs = length pTs ∧
                (∃ D' rT' maxl' ins'.
                    method (G, D) sig0 = Some(D', rT', (maxl', ins')) ∧
                    G ⊢ rT0 ≤ rT')) ∧
                correct_frame G hp (tl ST, LT) maxl ins f ∧
                correct_frames G hp phi rT sig frs))))"

```

constdefs

```

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

"correct_state G phi == λ(xp, hp, frs).
    case xp of

```

```

None => (case frs of
  [] => True
| (f#fs) => G⊢h hp√ ∧
  (let (stk,loc,C,sig,pc) = f
  in
  ∃ rT maxl ins s.
  method (G,C) sig = Some(C,rT,(maxl,ins)) ∧
  phi C sig ! pc = Some s ∧
  correct_frame G hp s maxl ins f ∧
  correct_frames G hp phi rT sig fs))
| Some x => True"

syntax (HTML)
correct_state :: "[jvm_prog,prog_type,jvm_state] => bool"
               ("_,_ |-JVM _ [ok]" [51,51] 50)

lemma sup_heap_newref:
  "hp x = None ==> hp ≤| hp(newref hp ↦ obj)"
apply (unfold hext_def)
apply clarsimp
apply (drule newref_None 1) back
apply simp
done

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

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

lemma approx_val_Null:
  "approx_val G hp Null (Ok (RefT x))"
by (auto intro: null simp add: approx_val_def)

lemma approx_val_imp_approx_val_assConvertible [rule_format]:
  "wf_prog wt G ==> approx_val G hp v (Ok T) --> G⊢ T ≤ T'
  --> approx_val G hp v (Ok T')"
by (cases T) (auto intro: conf_widen simp add: approx_val_def)

lemma approx_val_imp_approx_val_sup_heap [rule_format]:
  "approx_val G hp v at --> hp ≤| hp' --> approx_val G hp' v at"
apply (simp add: approx_val_def split: err.split)
apply (blast intro: conf_hext)
done

```

```

lemma approx_val_imp_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 v us) --> (G ⊢ us ≤o us')
  --> (approx_val G h v us')"
apply (simp add: sup PTS_eq approx_val_def split: err.split)
apply (blast intro: conf_widen)
done

```

```

lemma approx_loc_imp_approx_val_sup:
  "[| wf_prog wt G; approx_loc G hp loc LT; idx < length LT;
    v = loc!idx; G ⊢ LT!idx ≤o at |]
  ==> approx_val G hp 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

```

```

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

```

```

lemma assConv_approx_stk_imp_approx_loc [rule_format]:
  "wf_prog wt G ==> (∀ tt' ∈ set (zip tys_n ts). tt' ∈ widen G)
  --> length tys_n = length ts --> approx_stk G hp s tys_n -->
  approx_loc G hp 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_imp_approx_val_assConvertible)
apply auto
done

```

```

lemma approx_loc_imp_approx_loc_sup_heap [rule_format]:
  "∀ lvars. approx_loc G hp lvars lt --> hp ≤| hp'
  --> approx_loc G hp' lvars lt"
apply (unfold approx_loc_def list_all2_def)
apply (cases lt)
  apply simp
  apply clarsimp
  apply (rule approx_val_imp_approx_val_sup_heap)
  apply auto
done

```

```

lemma approx_loc_imp_approx_loc_sup [rule_format]:
  "wf_prog wt G ==> approx_loc G hp lvars lt --> G ⊢ lt ≤ 1 lt'
  --> approx_loc G hp lvars lt'"
apply (unfold 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 loc LT) --> (approx_val G hp x X)
  --> (approx_loc G hp (loc[idx:=x]) (LT[idx:=X]))"
apply (unfold approx_loc_def list_all2_def)
apply (auto dest: subsetD [OF set_update_subset_insert] simp add: zip_update)
done

lemmas [cong] = conj_cong

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

lemmas [cong del] = conj_cong

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

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

lemma approx_stk_imp_approx_stk_sup_heap [rule_format]:
  "∀ lvars. approx_stk G hp lvars lt --> hp ≤ | hp'
  --> approx_stk G hp' 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 lvars st --> (G ⊢ map Ok st ≤ 1 (map Ok st'))
  --> approx_stk G hp 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 [] []"
```

```
by (simp add: approx_stk_def approx_loc_def)
```

```
lemma approx_stk_Cons [iff]:
```

```
"approx_stk G hp (x # stk) (S#ST) ="
```

```
(approx_val G hp x (Ok S) ∧ approx_stk G hp stk ST)"
```

```
by (simp add: approx_stk_def approx_loc_def)
```

```
lemma approx_stk_Cons_lemma [iff]:
```

```
"approx_stk G hp stk (S#ST') ="
```

```
(∃ s stk'. stk = s#stk' ∧ approx_val G hp s (Ok S) ∧ approx_stk G hp stk' ST')"
```

```
by (simp add: list_all2_Cons2 approx_stk_def approx_loc_def)
```

```
lemma approx_stk_append_lemma:
```

```
"approx_stk G hp stk (S@ST') ==>
```

```
(∃ s stk'. stk = s@stk' ∧ length s = length S ∧ length stk' = length ST' ∧  
approx_stk G hp s S ∧ approx_stk G hp stk' ST')"
```

```
by (simp add: list_all2_append2 approx_stk_def approx_loc_def)
```

```
lemma correct_init_obj:
```

```
"[|is_class G C; wf_prog wt G|] ==>
```

```
G,h ⊢ (C, map_of (map (λ(f,fT).(f,default_val fT)) (fields(G,C)))) √"
```

```
apply (unfold oconf_def lconf_def)
```

```
apply (simp add: map_of_map)
```

```
apply (force intro: defval_conf dest: map_of_SomeD is_type_fields)
```

```
done
```

```
lemma oconf_imp_oconf_field_update [rule_format]:
```

```
"[|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 x = None --> G,hp ⊢ obj √ --> G,hp ⊢ obj' √ --> G,(hp(newref hp ⊢ 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_imp_approx_val_heap_update)
done

```

```

lemma hconf_imp_hconf_newref [rule_format]:
  "hp x = None --> G ⊢h hp √ --> G, hp ⊢h obj √ --> G ⊢h hp (newref hp ↦ 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_imp_oconf_field_update
  simp add: obj_ty_def)
done

```

```

lemmas [simp del] = fun_upd_apply

```

```

lemma correct_frames_imp_correct_frames_field_update [rule_format]:
  "∀ rT C sig. correct_frames G hp phi rT sig 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)))) phi rT sig frs"
apply (induct frs)
  apply simp
  apply (clarsimp simp add: correct_frame_def)
  apply (intro exI conjI)
    apply simp
    apply simp
    apply force
    apply simp
  apply (rule approx_stk_imp_approx_stk_sup_heap)
    apply simp
  apply (rule sup_heap_update_value)
    apply simp
  apply (rule approx_loc_imp_approx_loc_sup_heap)
    apply simp
  apply (rule sup_heap_update_value)
  apply simp
done

```

```

lemma correct_frames_imp_correct_frames_newref [rule_format]:
  "∀ rT C sig. hp x = None --> correct_frames G hp phi rT sig frs ∧ oconf G hp obj
  --> correct_frames G (hp(newref hp ↦ obj)) phi rT sig frs"

```

```
apply (induct frs)
  apply simp
apply (clarsimp simp add: correct_frame_def)
apply (intro exI conjI)
  apply simp
  apply simp
  apply force
  apply simp
apply (rule approx_stk_imp_approx_stk_sup_heap)
  apply simp
apply (rule sup_heap_newref)
  apply simp
apply (rule approx_loc_imp_approx_loc_sup_heap)
  apply simp
apply (rule sup_heap_newref)
  apply simp
done

end
```

23 BV Type Safety Proof

theory BVSpecTypeSafe = Correct:

lemmas defs1 = sup_state_def correct_state_def correct_frame_def
wt_instr_def step_def

lemmas [simp del] = split_paired_All

lemma wt_jvm_prog_impl_wt_instr_cor:

"[| wt_jvm_prog G phi; method (G,C) sig = Some (C,rT,maxl,ins);
G,phi ⊢ JVM (None, hp, (stk,loc,C,sig,pc)#frs) √ |]
==> wt_instr (ins!pc) G rT (phi C sig) (length ins) pc"

apply (unfold correct_state_def Let_def correct_frame_def)

apply simp

apply (blast intro: wt_jvm_prog_impl_wt_instr)

done

lemma Load_correct:

"[| wf_prog wt G;
method (G,C) sig = Some (C,rT,maxl,ins);
ins!pc = Load idx;
wt_instr (ins!pc) G rT (phi C sig) (length ins) pc;
Some state' = exec (G, None, hp, (stk,loc,C,sig,pc)#frs);
G,phi ⊢ JVM (None, hp, (stk,loc,C,sig,pc)#frs) √ |]
==> G,phi ⊢ JVM state' √"

apply (clarsimp simp add: defs1 map_eq_Cons)

apply (rule conjI)

apply (rule approx_loc_imp_approx_val_sup)

apply simp+

apply (blast intro: approx_stk_imp_approx_stk_sup
approx_loc_imp_approx_loc_sup)

done

lemma Store_correct:

"[| wf_prog wt G;
method (G,C) sig = Some (C,rT,maxl,ins);
ins!pc = Store idx;
wt_instr (ins!pc) G rT (phi C sig) (length ins) pc;
Some state' = exec (G, None, hp, (stk,loc,C,sig,pc)#frs);
G,phi ⊢ JVM (None, hp, (stk,loc,C,sig,pc)#frs) √ |]
==> G,phi ⊢ JVM state' √"

apply (clarsimp simp add: defs1 map_eq_Cons)

apply (rule conjI)

apply (blast intro: approx_stk_imp_approx_stk_sup)

apply (blast intro: approx_loc_imp_approx_loc_subst
approx_loc_imp_approx_loc_sup)

done

lemma conf_Intg_Integer [iff]:

"G,h ⊢ Intg i :: ≤ PrimT Integer"

by (simp add: conf_def)

lemma Bipush_correct:

```
"[| wf_prog wt G;
  method (G,C) sig = Some (C,rT,maxl,ins);
  ins!pc = Bipush i;
  wt_instr (ins!pc) G rT (phi C sig) (length ins) pc;
  Some state' = exec (G, None, hp, (stk,loc,C,sig,pc)#frs);
  G,phi ⊢JVM (None, hp, (stk,loc,C,sig,pc)#frs) √ |]
==> G,phi ⊢JVM state' √"
apply (clarsimp simp add: defs1 approx_val_def sup PTS_eq map_eq_Cons)
apply (blast intro: approx_stk_imp_approx_stk_sup
               approx_loc_imp_approx_loc_sup)
done
```

lemma NT_subtype_conv:

" $G ⊢ NT ⊆ T = (T=NT ∨ (∃ C. T = \text{Class } C))$ "

proof -

have "!!T T'. $G ⊢ T' ⊆ T ==> T' = NT --> (T=NT ∨ (∃ C. T = \text{Class } C))$ "

apply (erule widen.induct)

apply auto

apply (case_tac R)

apply auto

done

note 1 = this

show ?thesis

by (force intro: null dest: 1)

qed

lemma Aconst_null_correct:

```
"[| wf_prog wt G;
  method (G,C) sig = Some (C,rT,maxl,ins);
  ins!pc = Aconst_null;
  wt_instr (ins!pc) G rT (phi C sig) (length ins) pc;
  Some state' = exec (G, None, hp, (stk,loc,C,sig,pc)#frs);
  G,phi ⊢JVM (None, hp, (stk,loc,C,sig,pc)#frs) √ |]
==> G,phi ⊢JVM state' √"
apply (clarsimp simp add: defs1 map_eq_Cons)
apply (rule conjI)
  apply (force simp add: approx_val_Null NT_subtype_conv)
apply (blast intro: approx_stk_imp_approx_stk_sup
               approx_loc_imp_approx_loc_sup)
done
```

lemma Cast_conf2:

"[| wf_prog ok G; $G, h ⊢ v :: ⊆ \text{RefT } rt$; cast_ok G C h v;
 $G ⊢ \text{Class } C ⊆ T$; 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: null rtranc1_trans)
done

lemma Checkcast_correct:
"[| wf_prog wt G;
  method (G,C) sig = Some (C,rT,maxl,ins);
  ins!pc = Checkcast D;
  wt_instr (ins!pc) G rT (phi C sig) (length ins) pc;
  Some state' = exec (G, None, hp, (stk,loc,C,sig,pc)#frs) ;
  G,phi ⊢JVM (None, hp, (stk,loc,C,sig,pc)#frs) √ |]
==> G,phi ⊢JVM state' √"
apply (clarsimp simp add: defs1 map_eq_Cons raise_xcpt_def approx_val_def)
apply (blast intro: approx_stk_imp_approx_stk_sup
  approx_loc_imp_approx_loc_sup Cast_conf2)
done

```

```

lemma Getfield_correct:
"[| wf_prog wt G;
  method (G,C) sig = Some (C,rT,maxl,ins);
  ins!pc = Getfield F D;
  wt_instr (ins!pc) G rT (phi C sig) (length ins) pc;
  Some state' = exec (G, None, hp, (stk,loc,C,sig,pc)#frs) ;
  G,phi ⊢JVM (None, hp, (stk,loc,C,sig,pc)#frs) √ |]
==> G,phi ⊢JVM state' √"
apply (clarsimp simp add: defs1 raise_xcpt_def map_eq_Cons approx_val_def
  split: option.split)
apply (frule conf_widen)
apply assumption+
apply (drule conf_RefTD)
apply (clarsimp simp add: defs1 approx_val_def)
apply (rule conjI)
  apply (drule widen_cfs_fields)
  apply assumption+
  apply (force intro: conf_widen simp add: hconf_def oconf_def lconf_def)
apply (blast intro: approx_stk_imp_approx_stk_sup
  approx_loc_imp_approx_loc_sup)
done

```

```

lemma Putfield_correct:
"[| wf_prog wt G;
  method (G,C) sig = Some (C,rT,maxl,ins);
  ins!pc = Putfield F D;
  wt_instr (ins!pc) G rT (phi C sig) (length ins) pc;
  Some state' = exec (G, None, hp, (stk,loc,C,sig,pc)#frs) ;
  G,phi ⊢JVM (None, hp, (stk,loc,C,sig,pc)#frs) √ |]

```

```

==> G,phi ⊢ JVM state' ✓"
apply (clarsimp simp add: defs1 raise_xcpt_def split: option.split List.split)
apply (clarsimp simp add: approx_val_def)
apply (frule conf_widen)
prefer 2
apply assumption
apply assumption
apply (drule conf_RefTD)
apply clarsimp
apply (blast
  intro:
    sup_heap_update_value 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 hconf_imp_hconf_field_update
    correct_frames_imp_correct_frames_field_update conf_widen
  dest:
    widen_cfs_fields)
done

lemma collapse_paired_All:
  "(∀ x y. P(x,y)) = (∀ z. P z)"
  by fast

lemma New_correct:
  "[| wf_prog wt G;
    method (G,C) sig = Some (C,rT,maxl,ins);
    ins!pc = New cl_idx;
    wt_instr (ins!pc) G rT (phi C sig) (length ins) pc;
    Some state' = exec (G, None, hp, (stk,loc,C,sig,pc)#frs) ;
    G,phi ⊢ JVM (None, hp, (stk,loc,C,sig,pc)#frs) ✓ |]
  ==> G,phi ⊢ JVM state' ✓"
apply (clarsimp simp add: NT_subtype_conv approx_val_def conf_def defs1
  fun_upd_apply map_eq_Cons is_class_def raise_xcpt_def init_vars_def
  split: option.split)
apply (force dest!: iffD1 [OF collapse_paired_All]
  intro: sup_heap_newref 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
    hconf_imp_hconf_newref correct_frames_imp_correct_frames_newref
    correct_init_obj
  simp add: NT_subtype_conv approx_val_def conf_def defs1
  fun_upd_apply map_eq_Cons is_class_def raise_xcpt_def init_vars_def
  split: option.split)
done

lemmas [simp del] = split_paired_Ex

lemma Invoke_correct:

```

```

"/ wt_jvm_prog G phi;
method (G,C) sig = Some (C,rT,maxl,ins);
ins ! pc = Invoke C' mn pTs;
wt_instr (ins!pc) G rT (phi C sig) (length ins) pc;
Some state' = exec (G, None, hp, (stk,loc,C,sig,pc)#frs) ;
G,phi ⊢JVM (None, hp, (stk,loc,C,sig,pc)#frs)√ /]
==> G,phi ⊢JVM state'√"
proof -
  assume wtprog: "wt_jvm_prog G phi"
  assume method: "method (G,C) sig = Some (C,rT,maxl,ins)"
  assume ins: "ins ! pc = Invoke C' mn pTs"
  assume wti: "wt_instr (ins!pc) G rT (phi C sig) (length ins) pc"
  assume state': "Some state' = exec (G, None, hp, (stk,loc,C,sig,pc)#frs)"
  assume approx: "G,phi ⊢JVM (None, hp, (stk,loc,C,sig,pc)#frs)√"

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

  from ins method approx
  obtain s where
    heap_ok: "G⊢h hp√" and
    phi_pc: "phi C sig!pc = Some s" and
    frame: "correct_frame G hp s maxl ins (stk, loc, C, sig, pc)" and
    frames: "correct_frames G hp phi rT sig frs"
    by (clarsimp simp add: correct_state_def)

  from ins wti phi_pc
  obtain apTs X ST LT D' rT body where
    s: "s = (rev apTs @ X # ST, LT)" and
    l: "length apTs = length pTs" and
    X: "G⊢ X ⊆ Class C'" and
    w: "∀x∈set (zip apTs pTs). x ∈ widen G" and
    mC': "method (G, C') (mn, pTs) = Some (D', rT, body)" and
    pc: "Suc pc < length ins" and
    step: "G ⊢ step (Invoke C' mn pTs) G (Some s) <= phi C sig!Suc pc"
    by (simp add: wt_instr_def) (elim exE conjE, rule that)

  from step
  obtain ST' LT' where
    s': "phi C sig ! Suc pc = Some (ST', LT')"
    by (simp add: step_def split_paired_Ex) (elim, rule that)

  from X
  obtain T where
    X_Ref: "X = RefT T"
    by - (drule widen_RefT2, erule exE, rule that)

  from s ins frame
  obtain
    a_stk: "approx_stk G hp stk (rev apTs @ X # ST)" and

```

```

a_loc: "approx_loc G hp loc LT" and
suc_l: "length loc = Suc (length (snd sig) + maxl)"
by (simp add: correct_frame_def)

from a_stk
obtain opTs stk' oX where
  opTs: "approx_stk G hp opTs (rev apTs)" and
  oX: "approx_val G hp oX (Ok X)" and
  a_stk': "approx_stk G hp stk' ST" and
  stk': "stk = opTs @ oX # stk'" and
  l_o: "length opTs = length apTs"
      "length stk' = length ST"
  by - (drule approx_stk_append_lemma, simp, elim, simp)

from oX
have "∃ T'. typeof (option_map obj_ty ∘ hp) oX = Some T' ∧ G ⊢ T' ≤ X"
  by (clarsimp simp add: approx_val_def conf_def)

with X_Ref
obtain T' where
  oX_Ref: "typeof (option_map obj_ty ∘ hp) oX = Some (RefT T'"
          "G ⊢ RefT T' ≤ X"
  apply (elim, simp)
  apply (frule widen_RefT2)
  by (elim, simp)

from stk' l_o l
have oX_pos: "last (take (Suc (length pTs)) stk) = oX"
  by simp

with state' method ins
have Null_ok: "oX = Null ==> ?thesis"
  by (simp add: correct_state_def raise_xcpt_def)

{ fix ref
  assume oX_Addr: "oX = Addr ref"

  with oX_Ref
  obtain obj where
    loc: "hp ref = Some obj" "obj_ty obj = RefT T'"
    by clarsimp

  then
  obtain D where
    obj_ty: "obj_ty obj = Class D"
    by (auto simp add: obj_ty_def)

  with X_Ref oX_Ref loc
  obtain D: "G ⊢ Class D ≤ X"
    by simp

  with X_Ref

```

```

obtain X' where
  X': "X = Class X'"
  by - (drule widen_Class, elim, rule that)

with X
have "G ⊢ X' ≤C C'"
  by simp

with mC' wfprog
obtain D0 rT0 maxl0 ins0 where
  mX: "method (G, X') (mn, pTs) = Some (D0, rT0, maxl0, ins0)" "G ⊢ rT0 ≤rT"
  by (auto dest: subtype_widen_methd)

from X' D
have "G ⊢ D ≤C X'"
  by simp

with wfprog mX
obtain D'' rT' mxl' ins' where
  mD: "method (G, D) (mn, pTs) = Some (D'', rT', mxl', ins')"
      "G ⊢ rT' ≤rT rT0"
  by (auto dest: subtype_widen_methd)

from mX mD
have rT': "G ⊢ rT' ≤rT rT"
  by - (rule widen_trans)

from mD wfprog
obtain mD'':
  "method (G, D'') (mn, pTs) = Some (D'', rT', mxl', ins')"
  "is_class G D''"
  by (auto dest: method_in_md)

from loc obj_ty
have "fst (the (hp ref)) = D"
  by (simp add: obj_ty_def)

with oX_Addr oX_pos state' method ins stk' l_o l loc obj_ty mD
have state'_val:
  "state' =
    Norm (hp, ([], Addr ref # rev opTs @ replicate mxl' arbitrary,
      D'', (mn, pTs), 0) # (stk', loc, C, sig, Suc pc) # frs)"
  (is "state' = Norm (hp, ?f # ?f' # frs)")
  by (simp add: raise_xcpt_def)

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

then
obtain LT0 where
  LT0: "phi D'' (mn, pTs) ! 0 = Some ([], LT0)"

```

```

by (clarsimp simp add: wt_start_def sup_state_def)

have c_f: "correct_frame G hp ([], LT0) mxl' ins' ?f"
proof -
  from start LT0
  have sup_loc:
    "G ⊢ (Ok (Class D'')) # map Ok pTs @ replicate mxl' Err) ≤1 LT0"
    (is "G ⊢ ?LT ≤1 LT0")
  by (simp add: wt_start_def sup_state_def)

  have r: "approx_loc G hp (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
  have "G ⊢ Class D ≤1 Class D'"
    by (auto dest: method_wf_mdecl)
  with obj_ty loc
  have a: "approx_val G hp (Addr ref) (Ok (Class D'))"
    by (simp add: approx_val_def conf_def)

  from w l
  have "∀x∈set (zip (rev apTs) (rev pTs)). x ∈ widen G"
    by (auto simp add: zip_rev)
  with wfprog l l_o opTs
  have "approx_loc G hp opTs (map Ok (rev pTs))"
    by (auto intro: assConv_approx_stk_imp_approx_loc)
  hence "approx_stk G hp opTs (rev pTs)"
    by (simp add: approx_stk_def)
  hence "approx_stk G hp (rev opTs) pTs"
    by (simp add: approx_stk_rev)

  with r a l_o l
  have "approx_loc G hp (Addr ref # rev opTs @ replicate mxl' arbitrary) ?LT"
    (is "approx_loc G hp ?lt ?LT")
    by (auto simp add: approx_loc_append approx_stk_def)

  from wfprog this sup_loc
  have "approx_loc G hp ?lt LT0"
    by (rule approx_loc_imp_approx_loc_sup)

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

have c_f': "correct_frame G hp (tl ST', LT') maxl ins' ?f'"
proof -
  from s s' mC' step l
  have "G ⊢ LT ≤1 LT'"
    by (simp add: step_def sup_state_def)
  with wfprog a_loc

```

```

    have a: "approx_loc G hp loc LT'"
      by (rule approx_loc_imp_approx_loc_sup)
    moreover
    from s s' mC' step 1
    have "G ⊢ map Ok ST ≤ map Ok (tl ST')"
      by (auto simp add: step_def sup_state_def map_eq_Cons)
    with wfprog a_stk'
    have "approx_stk G hp stk' (tl ST')"
      by (rule approx_stk_imp_approx_stk_sup)
    ultimately
    show ?thesis
      by (simp add: correct_frame_def suc_1 pc)
  qed

  with state'_val heap_ok mD'' ins method phi_pc s X' 1
    frames s' LTO c_f c_f'
  have ?thesis
    by (simp add: correct_state_def) (intro exI conjI, force+)
}

with Null_ok oX_Ref
show "G, phi ⊢ JVM state' √"
  by - (cases oX, auto)
qed

lemmas [simp del] = map_append

lemma Return_correct:
  "[| wt_jvm_prog G phi;
    method (G,C) sig = Some (C,rT,maxl,ins);
    ins ! pc = Return;
    wt_instr (ins!pc) G rT (phi C sig) (length ins) pc;
    Some state' = exec (G, None, hp, (stk,loc,C,sig,pc)#frs) ;
    G, phi ⊢ JVM (None, hp, (stk,loc,C,sig,pc)#frs) √ |]
  ==> G, phi ⊢ JVM state' √"
  apply (clarsimp simp add: neq_Nil_conv defs1 split: split_if_asm)
  apply (frule wt_jvm_prog_impl_wt_instr)
  apply (assumption, erule Suc_lessD)
  apply (unfold wt_jvm_prog_def)
  apply (fastsimp
    dest: subcls_widen_methd
    elim: widen_trans [COMP swap_premis_rl]
    intro: conf_widen
    simp: approx_val_def append_eq_conv_conj map_eq_Cons defs1)
done

lemmas [simp] = map_append

lemma Goto_correct:
  "[| wf_prog wt G;
    method (G,C) sig = Some (C,rT,maxl,ins);
    ins ! pc = Goto branch;

```

```

    wt_instr (ins!pc) G rT (phi C sig) (length ins) pc;
    Some state' = exec (G, None, hp, (stk,loc,C,sig,pc)#frs) ;
    G,phi ⊢JVM (None, hp, (stk,loc,C,sig,pc)#frs)√ []
==> G,phi ⊢JVM state'√"
apply (clarsimp simp add: defs1)
apply (fast intro: approx_loc_imp_approx_loc_sup
        approx_stk_imp_approx_stk_sup)
done

```

lemma Ifcmpeq_correct:

```

"[] wf_prog wt G;
  method (G,C) sig = Some (C,rT,maxl,ins);
  ins ! pc = Ifcmpeq branch;
  wt_instr (ins!pc) G rT (phi C sig) (length ins) pc;
  Some state' = exec (G, None, hp, (stk,loc,C,sig,pc)#frs) ;
  G,phi ⊢JVM (None, hp, (stk,loc,C,sig,pc)#frs)√ []
==> G,phi ⊢JVM state'√"
apply (clarsimp simp add: defs1)
apply (fast intro: approx_loc_imp_approx_loc_sup
        approx_stk_imp_approx_stk_sup)
done

```

lemma Pop_correct:

```

"[] wf_prog wt G;
  method (G,C) sig = Some (C,rT,maxl,ins);
  ins ! pc = Pop;
  wt_instr (ins!pc) G rT (phi C sig) (length ins) pc;
  Some state' = exec (G, None, hp, (stk,loc,C,sig,pc)#frs) ;
  G,phi ⊢JVM (None, hp, (stk,loc,C,sig,pc)#frs)√ []
==> G,phi ⊢JVM state'√"
apply (clarsimp simp add: defs1)
apply (fast intro: approx_loc_imp_approx_loc_sup
        approx_stk_imp_approx_stk_sup)
done

```

lemma Dup_correct:

```

"[] wf_prog wt G;
  method (G,C) sig = Some (C,rT,maxl,ins);
  ins ! pc = Dup;
  wt_instr (ins!pc) G rT (phi C sig) (length ins) pc;
  Some state' = exec (G, None, hp, (stk,loc,C,sig,pc)#frs) ;
  G,phi ⊢JVM (None, hp, (stk,loc,C,sig,pc)#frs)√ []
==> G,phi ⊢JVM state'√"
apply (clarsimp simp add: defs1 map_eq_Cons)
apply (force intro: approx_stk_imp_approx_stk_sup
        approx_val_imp_approx_val_sup
        approx_loc_imp_approx_loc_sup
        simp add: defs1 map_eq_Cons)
done

```

lemma Dup_x1_correct:

```

"/ wf_prog wt G;
  method (G,C) sig = Some (C,rT,maxl,ins);
  ins ! pc = Dup_x1;
  wt_instr (ins!pc) G rT (phi C sig) (length ins) pc;
  Some state' = exec (G, None, hp, (stk,loc,C,sig,pc)#frs) ;
  G,phi ⊢JVM (None, hp, (stk,loc,C,sig,pc)#frs)√ /]
==> G,phi ⊢JVM state'√"
apply (clarsimp simp add: defs1 map_eq_Cons)
apply (force intro: approx_stk_imp_approx_stk_sup
                  approx_val_imp_approx_val_sup
                  approx_loc_imp_approx_loc_sup
                  simp add: defs1 map_eq_Cons)
done

```

lemma Dup_x2_correct:

```

"/ wf_prog wt G;
  method (G,C) sig = Some (C,rT,maxl,ins);
  ins ! pc = Dup_x2;
  wt_instr (ins!pc) G rT (phi C sig) (length ins) pc;
  Some state' = exec (G, None, hp, (stk,loc,C,sig,pc)#frs) ;
  G,phi ⊢JVM (None, hp, (stk,loc,C,sig,pc)#frs)√ /]
==> G,phi ⊢JVM state'√"
apply (clarsimp simp add: defs1 map_eq_Cons)
apply (force intro: approx_stk_imp_approx_stk_sup
                  approx_val_imp_approx_val_sup
                  approx_loc_imp_approx_loc_sup
                  simp add: defs1 map_eq_Cons)
done

```

lemma Swap_correct:

```

"/ wf_prog wt G;
  method (G,C) sig = Some (C,rT,maxl,ins);
  ins ! pc = Swap;
  wt_instr (ins!pc) G rT (phi C sig) (length ins) pc;
  Some state' = exec (G, None, hp, (stk,loc,C,sig,pc)#frs) ;
  G,phi ⊢JVM (None, hp, (stk,loc,C,sig,pc)#frs)√ /]
==> G,phi ⊢JVM state'√"
apply (clarsimp simp add: defs1 map_eq_Cons)
apply (force intro: approx_stk_imp_approx_stk_sup
                  approx_val_imp_approx_val_sup
                  approx_loc_imp_approx_loc_sup
                  simp add: defs1 map_eq_Cons)
done

```

lemma IAdd_correct:

```

"/ wf_prog wt G;
  method (G,C) sig = Some (C,rT,maxl,ins);
  ins ! pc = IAdd;
  wt_instr (ins!pc) G rT (phi C sig) (length ins) pc;
  Some state' = exec (G, None, hp, (stk,loc,C,sig,pc)#frs) ;
  G,phi ⊢JVM (None, hp, (stk,loc,C,sig,pc)#frs)√ /]
==> G,phi ⊢JVM state'√"

```

```

apply (clarsimp simp add: defs1 map_eq_Cons approx_val_def conf_def)
apply (blast intro: approx_stk_imp_approx_stk_sup
              approx_val_imp_approx_val_sup
              approx_loc_imp_approx_loc_sup)
done

```

```

lemma instr_correct:
  "[| wt_jvm_prog G phi;
    method (G,C) sig = Some (C,rT,maxl,ins);
    Some state' = exec (G, None, hp, (stk,loc,C,sig,pc)#frs);
    G,phi ⊢JVM (None, hp, (stk,loc,C,sig,pc)#frs)√ |]
  ==> G,phi ⊢JVM state'√"
  apply (frule wt_jvm_prog_impl_wt_instr_cor)
  apply assumption+
  apply (cases "ins!pc")
  prefer 9
  apply (blast intro: Invoke_correct)
  prefer 9
  apply (blast intro: Return_correct)
  apply (unfold wt_jvm_prog_def)
  apply (fast intro:
    Load_correct Store_correct Bipush_correct Aconst_null_correct
    Checkcast_correct Getfield_correct Putfield_correct New_correct
    Goto_correct Ifcmpeq_correct Pop_correct Dup_correct
    Dup_x1_correct Dup_x2_correct Swap_correct IAdd_correct)+
done

```

```

lemma correct_state_impl_Some_method:
  "G,phi ⊢JVM (None, hp, (stk,loc,C,sig,pc)#frs)√
  ==> ∃meth. method (G,C) sig = Some(C,meth)"
  by (auto simp add: correct_state_def Let_def)

```

```

lemma BV_correct_1 [rule_format]:
  "!!state. [| wt_jvm_prog G phi; G,phi ⊢JVM state√|]
  ==> exec (G,state) = Some state' --> G,phi ⊢JVM state'√"
  apply (simp only: split_tupled_all)
  apply (rename_tac xp hp frs)
  apply (case_tac xp)
  apply (case_tac frs)
  apply simp
  apply (simp only: split_tupled_all)
  apply hypsubst
  apply (frule correct_state_impl_Some_method)
  apply (force intro: instr_correct)
  apply (case_tac frs)

```

```

apply simp_all
done

```

```

lemma L0:

```

```

  "[| xp=None; frs≠[] |] ==> (∃ state'. exec (G,xp,hp,frs) = Some state')"
  by (clarsimp simp add: neq_Nil_conv simp del: split_paired_Ex)

```

```

lemma L1:

```

```

  "[|wt_jvm_prog G phi; G,phi ⊢JVM (xp,hp,frs)√; xp=None; frs≠[]|]
  ==> ∃ state'. exec(G,xp,hp,frs) = Some state' ∧ G,phi ⊢JVM state'√"
  apply (drule L0)
  apply assumption
  apply (fast intro: BV_correct_1)
done

```

```

theorem BV_correct [rule_format]:

```

```

  "[| wt_jvm_prog G phi; G ⊢ s -jvm-> t |] ==> G,phi ⊢JVM s√ --> G,phi ⊢JVM t√"
  apply (unfold exec_all_def)
  apply (erule rtrancI_induct)
  apply simp
  apply (auto intro: BV_correct_1)
done

```

```

theorem BV_correct_initial:

```

```

  "[| wt_jvm_prog G phi;
    G ⊢ s0 -jvm-> (None,hp,(stk,loc,C,sig,pc)#frs); G,phi ⊢JVM s0 √ |]
  ==> approx_stk G hp stk (fst (the ((phi C) sig) ! pc))) ∧
    approx_loc G hp loc (snd (the ((phi C) sig) ! pc)))"
  apply (drule BV_correct)
  apply assumption+
  apply (simp add: correct_state_def correct_frame_def split_def
    split: option.splits)
done

```

```

end

```

24 The Lightweight Bytecode Verifier

theory LBVSPEC = Step :

types

```
certificate      = "state_type option list"
class_certificate = "sig => certificate"
prog_certificate = "cname => class_certificate"
```

constdefs

```
check_cert :: "[instr, jvm_prog, state_type option, certificate, p_count, p_count]
=> bool"

"check_cert i G s cert pc max_pc ==  $\forall pc' \in \text{set } (\text{succs } i \text{ } pc). pc' < \text{max\_pc} \wedge$ 
  ( $pc' \neq pc+1 \rightarrow G \vdash \text{step } i \text{ } G \text{ } s \leq' \text{cert!pc}'$ )"
```

```
wtl_inst :: "[instr, jvm_prog, ty, state_type option, certificate, p_count, p_count]
=> state_type option err"

"wtl_inst i G rT s cert max_pc pc ==
  if app i G rT s  $\wedge$  check_cert i G s cert pc max_pc then
    if pc+1 mem (succs i pc) then Ok (step i G s) else Ok (cert!(pc+1))
  else Err"
```

constdefs

```
wtl_cert :: "[instr, jvm_prog, ty, state_type option, certificate, p_count, p_count]
=> state_type option err"

"wtl_cert i G rT s cert max_pc pc ==
  case cert!pc of
    None    => wtl_inst i G rT s cert max_pc pc
  | Some s' => if G  $\vdash$  s  $\leq'$  (Some s') then
    wtl_inst i G rT (Some s') cert max_pc pc
  else Err"
```

consts

```
wtl_inst_list :: "[instr list, jvm_prog, ty, certificate, p_count, p_count,
  state_type option] => state_type option err"
```

primrec

```
"wtl_inst_list []      G rT cert max_pc pc s = Ok s"
"wtl_inst_list (i#is) G rT cert max_pc pc s =
  (let s' = wtl_cert i G rT s cert max_pc pc in
   strict (wtl_inst_list is G rT cert max_pc (pc+1)) s')"
```

constdefs

```
wtl_method :: "[jvm_prog, cname, ty list, ty, nat, instr list, certificate] => bool"

"wtl_method G C pTs rT mxl ins cert ==
  let max_pc = length ins
in
  0 < max_pc  $\wedge$ 
  wtl_inst_list ins G rT cert max_pc 0
    (Some ([], (Ok (Class C))#((map Ok pTs))@(replicate mxl Err)))  $\neq$  Err"
```

```

wtl_jvm_prog :: "[jvm_prog, prog_certificate] => bool"
"wtl_jvm_prog G cert ==
  wf_prog (λG C (sig, rT, maxl, b). wtl_method G C (snd sig) rT maxl b (cert C sig)) G"

```

```

lemma wtl_inst_0k:
"(wtl_inst i G rT s cert max_pc pc = 0k s') =
  (app i G rT s ∧ (∀ pc' ∈ set (succs i pc).
    pc' < max_pc ∧ (pc' ≠ pc+1 --> G ⊢ step i G s <= cert!pc')) ∧
  (if pc+1 ∈ set (succs i pc) then s' = step i G s else s' = cert!(pc+1)))"
  by (auto simp add: wtl_inst_def check_cert_def set_mem_eq)

```

```

lemma strict_Some [simp]:
"(strict f x = 0k y) = (∃ z. x = 0k z ∧ f z = 0k y)"
  by (cases x, auto)

```

```

lemma wtl_Cons:
"wtl_inst_list (i#is) G rT cert max_pc pc s ≠ Err =
  (∃ s'. wtl_cert i G rT s cert max_pc pc = 0k s' ∧
    wtl_inst_list is G rT cert max_pc (pc+1) s' ≠ Err)"
  by (auto simp del: split_paired_Ex)

```

```

lemma wtl_append:
"∀ s pc. (wtl_inst_list (a@b) G rT cert mpc pc s = 0k s') =
  (∃ s''. wtl_inst_list a G rT cert mpc pc s = 0k s'' ∧
    wtl_inst_list b G rT cert mpc (pc+length a) s'' = 0k s')"
  (is "∀ s pc. ?C s pc a" is "?P a")
proof (induct ?P a)

```

```

  show "?P []" by simp

```

```

  fix x xs
  assume IH: "?P xs"

```

```

  show "?P (x#xs)"

```

```

  proof (intro allI)

```

```

    fix s pc

```

```

    show "?C s pc (x#xs)"

```

```

    proof (cases "wtl_cert x G rT s cert mpc pc")

```

```

      case Err thus ?thesis by simp

```

```

    next

```

```

      fix s0

```

```

      assume 0k: "wtl_cert x G rT s cert mpc pc = 0k s0"

```

```

      with IH

```

```

      have "?C s0 (Suc pc) xs" by blast

```

```

      with 0k

```

```

      show ?thesis by simp

```

```

    qed

```

qed
qed

lemma wtl_take:

"wtl_inst_list is G rT cert mpc pc s = Ok s'" ==>
 $\exists s'. \text{wtl_inst_list (take pc' is) G rT cert mpc pc s = Ok s'}$ "

proof -

assume "wtl_inst_list is G rT cert mpc pc s = Ok s'"

hence "wtl_inst_list (take pc' is @ drop pc' is) G rT cert mpc pc s = Ok s'"
 by simp

thus ?thesis

by (auto simp add: wtl_append simp del: append_take_drop_id)

qed

lemma take_Suc:

" $\forall n. n < \text{length } l \rightarrow \text{take (Suc } n) l = (\text{take } n l) @ [l!n]$ " (is "?P l")

proof (induct l)

show "?P []" by simp

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 wtl_Suc:

"[| wtl_inst_list (take pc is) G rT cert (length is) 0 s = Ok s';

wtl_cert (is!pc) G rT s' cert (length is) pc = Ok s'';

Suc pc < length is |] ==>

wtl_inst_list (take (Suc pc) is) G rT cert (length is) 0 s = Ok s'"

proof -

assume wtt: "wtl_inst_list (take pc is) G rT cert (length is) 0 s = Ok s'"

assume wtc: "wtl_cert (is!pc) G rT s' cert (length is) pc = Ok s'"

assume pc: "Suc pc < length is"

hence "take (Suc pc) is = (take pc is)@[is!pc]"

by (simp add: take_Suc)

with wtt wtc pc

show ?thesis

by (simp add: wtl_append min_def)

qed

```

lemma wtl_all:
  "[| wtl_inst_list is G rT cert (length is) 0 s ≠ Err;
    pc < length is |] ==>
    ∃ s' s''. wtl_inst_list (take pc is) G rT cert (length is) 0 s = Ok s' ∧
      wtl_cert (is!pc) G rT s' cert (length is) pc = Ok s'' "
proof -
  assume all: "wtl_inst_list is G rT cert (length is) 0 s ≠ Err"

  assume pc: "pc < length is"
  hence "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_inst_list (take pc is@i#r) G rT cert (length is) 0 s ≠ Err"
    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: wtl_append min_def)
qed

lemma unique_set:
  "(a,b,c,d)∈set l --> unique l --> (a',b',c',d') ∈ set l -->
    a = a' --> b=b' ∧ c=c' ∧ d=d'"
  by (induct "l") auto

lemma unique_epsilon:
  "(a,b,c,d)∈set l --> unique l -->
    (SOME (a',b',c',d')). (a',b',c',d') ∈ set l ∧ a'=a) = (a,b,c,d)"
  by (auto simp add: unique_set)

end

```

25 Correctness of the LBV

theory LBVCorrect = BVSpec + LBVSpec:

lemmas [simp del] = split_paired_Ex split_paired_All

constdefs

```
fits :: "[method_type, instr list, jvm_prog, ty, state_type option, certificate] => bool"
"fits phi is G rT s0 cert ==
  (∀ pc s1. pc < length is -->
    (wtl_inst_list (take pc is) G rT cert (length is) 0 s0 = Ok s1 -->
      (case cert!pc of None => phi!pc = s1
        | Some t => phi!pc = Some t)))"
```

constdefs

```
make_phi :: "[instr list, jvm_prog, ty, state_type option, certificate] => method_type"
"make_phi is G rT s0 cert ==
  map (λpc. case cert!pc of
    None => val (wtl_inst_list (take pc is) G rT cert (length is) 0 s0)
  | Some t => Some t) [0..length is()]"
```

lemma fitsD_None:

```
"[/fits phi is G rT s0 cert; pc < length is;
  wtl_inst_list (take pc is) G rT cert (length is) 0 s0 = Ok s1;
  cert ! pc = None/] ==> phi!pc = s1"
by (auto simp add: fits_def)
```

lemma fitsD_Some:

```
"[/fits phi is G rT s0 cert; pc < length is;
  wtl_inst_list (take pc is) G rT cert (length is) 0 s0 = Ok s1;
  cert ! pc = Some t/] ==> phi!pc = Some t"
by (auto simp add: fits_def)
```

lemma make_phi_Some:

```
"[/ pc < length is; cert!pc = Some t /] ==>
  make_phi is G rT s0 cert ! pc = Some t"
by (simp add: make_phi_def)
```

lemma make_phi_None:

```
"[/ pc < length is; cert!pc = None /] ==>
  make_phi is G rT s0 cert ! pc =
  val (wtl_inst_list (take pc is) G rT cert (length is) 0 s0)"
by (simp add: make_phi_def)
```

lemma exists_phi:

```
"∃ phi. fits phi is G rT s0 cert"
```

proof -

```
have "fits (make_phi is G rT s0 cert) is G rT s0 cert"
  by (auto simp add: fits_def make_phi_Some make_phi_None
    split: option.splits)
```

```

    thus ?thesis
    by blast
qed

```

```

lemma fits_lemma1:
  "[| wtl_inst_list is G rT cert (length is) 0 s = Ok s'; fits phi is G rT s cert |]
  ==> ∀pc t. pc < length is --> cert!pc = Some t --> phi!pc = Some t"
proof (intro strip)
  fix pc t
  assume "wtl_inst_list is G rT cert (length is) 0 s = Ok s'"
  then
  obtain s'' where
    "wtl_inst_list (take pc is) G rT cert (length is) 0 s = Ok s'"
    by (blast dest: wtl_take)
  moreover
  assume "fits phi is G rT s cert"
    "pc < length is"
    "cert ! pc = Some t"
  ultimately
  show "phi!pc = Some t" by (auto dest: fitsD_Some)
qed

```

```

lemma wtl_suc_pc:
  "[| wtl_inst_list is G rT cert (length is) 0 s ≠ Err;
    wtl_inst_list (take pc is) G rT cert (length is) 0 s = Ok s';
    wtl_cert (is!pc) G rT s' cert (length is) pc = Ok s'';
    fits phi is G rT s cert; Suc pc < length is |] ==>
  G ⊢ s'' <= phi ! Suc pc"
proof -
  assume all: "wtl_inst_list is G rT cert (length is) 0 s ≠ Err"
  assume fits: "fits phi is G rT s cert"

  assume wtl: "wtl_inst_list (take pc is) G rT cert (length is) 0 s = Ok s'" and
    wtc: "wtl_cert (is!pc) G rT s' cert (length is) pc = Ok s'" and
    pc: "Suc pc < length is"

  hence wts: "wtl_inst_list (take (Suc pc) is) G rT cert (length is) 0 s = Ok s'"
    by (rule wtl_Suc)

  from all
  have app:
    "wtl_inst_list (take (Suc pc) is@drop (Suc pc) is) G rT cert (length is) 0 s ≠ Err"
    by simp

  from pc
  have "0 < length (drop (Suc pc) is)"
    by simp
  then
  obtain l ls where
    "drop (Suc pc) is = l#ls"

```

```

    by (auto simp add: neq_Nil_conv simp del: length_drop)
  with app wts pc
  obtain x where
    "wtl_cert l G rT s'' cert (length is) (Suc pc) = Ok x"
  by (auto simp add: wtl_append min_def simp del: append_take_drop_id)

  hence c1: "!!t. cert!Suc pc = Some t ==> G ⊢ s'' <= cert!Suc pc"
  by (simp add: wtl_cert_def split: if_splits)
  moreover
  from fits pc wts
  have c2: "!!t. cert!Suc pc = Some t ==> phi!Suc pc = cert!Suc pc"
  by - (drule fitsD_Some, auto)
  moreover
  from fits pc wts
  have c3: "cert!Suc pc = None ==> phi!Suc pc = s'"
  by (rule fitsD_None)
  ultimately
  show ?thesis
  by - (cases "cert ! Suc pc", auto)
qed

```

```

lemma wtl_fits_wt:
  "[| wtl_inst_list is G rT cert (length is) 0 s ≠ Err;
    fits phi is G rT s cert; pc < length is |] ==>
    wt_instr (is!pc) G rT phi (length is) pc"
proof -
  assume fits: "fits phi is G rT s cert"

  assume pc: "pc < length is" and
    wtl: "wtl_inst_list is G rT cert (length is) 0 s ≠ Err"

  then
  obtain s' s'' where
    w: "wtl_inst_list (take pc is) G rT cert (length is) 0 s = Ok s'" and
    c: "wtl_cert (is!pc) G rT s' cert (length is) pc = Ok s'"
  by - (drule wtl_all, auto)

  from fits wtl pc
  have cert_Some:
    "!!t pc. [| pc < length is; cert!pc = Some t |] ==> phi!pc = Some t"
  by (auto dest: fits_lemma1)

  from fits wtl pc
  have cert_None: "cert!pc = None ==> phi!pc = s'"
  by - (drule fitsD_None)

  from pc c cert_None cert_Some
  have wti: "wtl_inst (is ! pc) G rT (phi!pc) cert (length is) pc = Ok s'"
  by (auto simp add: wtl_cert_def split: if_splits option.splits)

```

```

{ fix pc'
  assume pc': "pc' ∈ set (sucCs (is!pc) pc)"

  with wti
  have less: "pc' < length is"
    by (simp add: wtl_inst_0k)

  have "G ⊢ step (is!pc) G (phi!pc) <= phi ! pc'"
  proof (cases "pc' = Suc pc")
    case False
    with wti pc'
    have G: "G ⊢ step (is ! pc) G (phi ! pc) <= cert ! pc'"
      by (simp add: wtl_inst_0k)

    hence "cert!pc' = None ==> step (is ! pc) G (phi ! pc) = None"
      by simp
    hence "cert!pc' = None ==> ?thesis"
      by simp

  moreover
  { fix t
    assume "cert!pc' = Some t"
    with less
    have "phi!pc' = cert!pc'"
      by (simp add: cert_Some)
    with G
    have ?thesis
      by simp
  }

  ultimately
  show ?thesis by blast
next
case True
with pc' wti
have "step (is ! pc) G (phi ! pc) = s'"
  by (simp add: wtl_inst_0k)
also
from w c fits pc wtl
have "Suc pc < length is ==> G ⊢ s' <= phi ! Suc pc"
  by - (drule wtl_suc_pc)
with True less
have "G ⊢ s' <= phi ! Suc pc"
  by blast
finally
show ?thesis
  by (simp add: True)
qed
}

with wti

```

```

show ?thesis
  by (auto simp add: wtl_inst_0k wt_instr_def)
qed

```

```

lemma fits_first:
  "[| 0 < length is; wtl_inst_list is G rT cert (length is) 0 s ≠ Err;
    fits phi is G rT s cert |] ==>
  G ⊢ s <= ' phi ! 0"
proof -
  assume wtl: "wtl_inst_list is G rT cert (length is) 0 s ≠ Err"
  assume fits: "fits phi is G rT s cert"
  assume pc: "0 < length is"

  from wtl
  have wt0: "wtl_inst_list (take 0 is) G rT cert (length is) 0 s = 0k s"
    by simp

  with fits pc
  have "cert!0 = None ==> phi!0 = s"
    by (rule fitsD_None)
  moreover
  from fits pc wt0
  have "!!t. cert!0 = Some t ==> phi!0 = cert!0"
    by - (drule fitsD_Some, auto)
  moreover
  from pc
  obtain x xs where "is = x#xs"
    by (simp add: neq_Nil_conv) (elim, rule that)
  with wtl
  obtain s' where
    "wtl_cert x G rT s cert (length is) 0 = 0k s'"
    by simp (elim, rule that, simp)
  hence
    "!!t. cert!0 = Some t ==> G ⊢ s <= ' cert!0"
    by (simp add: wtl_cert_def split: if_splits)

  ultimately
  show ?thesis
    by - (cases "cert!0", auto)
qed

```

```

lemma wtl_method_correct:
  "wtl_method G C pTs rT mxl ins cert ==> ∃ phi. wt_method G C pTs rT mxl ins phi"
proof (unfold wtl_method_def, simp only: Let_def, elim conjE)
  let "?s0" = "Some ([], 0k (Class C) # map 0k pTs @ replicate mxl Err)"
  assume pc: "0 < length ins"
  assume wtl: "wtl_inst_list ins G rT cert (length ins) 0 ?s0 ≠ Err"

  obtain phi where fits: "fits phi ins G rT ?s0 cert"

```

```

    by (rule exists_phi [elim_format]) blast

with wtl
have allpc:
  "∀pc. pc < length ins --> wt_instr (ins ! pc) G rT phi (length ins) pc"
  by (blast intro: wtl_fits_wt)

from pc wtl fits
have "wt_start G C pTs mxl phi"
  by (unfold wt_start_def) (rule fits_first)

with pc allpc
show ?thesis by (auto simp add: wt_method_def)
qed

theorem wtl_correct:
  "wtl_jvm_prog G cert ==> ∃ Phi. wtl_jvm_prog G Phi"
proof (clarsimp simp add: wtl_jvm_prog_def wf_prog_def, intro conjI)

  assume wtl_prog: "wtl_jvm_prog G cert"
  thus "ObjectC ∈ set G" by (simp add: wtl_jvm_prog_def wf_prog_def)

from wtl_prog
show uniqueG: "unique G" by (simp add: wtl_jvm_prog_def wf_prog_def)

show "∃Phi. Ball (set G) (wf_cdecl (λG C (sig,rT,maxl,b).
  wt_method G C (snd sig) rT maxl b (Phi C sig)) G)"
  (is "∃Phi. ?Q Phi")
proof (intro exI)
  let "?Phi" = "λ C sig.
    let (C,x,y,mdecls) = SOME (Cl,x,y,mdecls). (Cl,x,y,mdecls) ∈ set G ∧ Cl = C;
    (sig,rT,maxl,b) = SOME (sg,rT,maxl,b). (sg,rT,maxl,b) ∈ set mdecls ∧ sg = sig
    in SOME phi. wt_method G C (snd sig) rT maxl b phi"
  from wtl_prog
  show "?Q ?Phi"
proof (unfold wf_cdecl_def, intro)
  fix x a b aa ba ab bb m
  assume 1: "x ∈ set G" "x = (a, b)" "b = (aa, ba)" "ba = (ab, bb)" "m ∈ set bb"
  with wtl_prog
  show "wf_mdecl (λG C (sig,rT,maxl,b).
    wt_method G C (snd sig) rT maxl b (?Phi C sig)) G a m"
  proof (simp add: wf_mdecl_def wtl_jvm_prog_def wf_prog_def wf_cdecl_def,
    elim conjE)
    apply_end (drule bspec, assumption, simp, elim conjE)
    assume "∀(sig,rT,mb)∈set bb. wf_mhead G sig rT ∧
      (λ(maxl,b). wt_method G a (snd sig) rT maxl b (cert a sig)) mb"
      "unique bb"
    with 1 uniqueG
    show "(λ(sig,rT,mb).
      wf_mhead G sig rT ∧
      (λ(maxl,b).

```

```

      wt_method G a (snd sig) rT maxl b
      ((λ(C,x,y,mdecls).
        (λ(sig,rT,maxl,b). Eps (wt_method G C (snd sig) rT maxl b))
        (SOME (sg,rT,maxl,b). (sg, rT, maxl, b) ∈ set mdecls ∧ sg = sig))
        (SOME (Cl,x,y,mdecls). (Cl, x, y, mdecls) ∈ set G ∧ Cl = a))) mb) m"
    by - (drule bspec, assumption,
      clarsimp dest!: wtl_method_correct,
      clarsimp intro!: someI simp add: unique_epsilon)
  qed
  qed (auto simp add: wtl_jvm_prog_def wf_prog_def wf_cdecl_def)
  qed
  qed

end

```

26 Monotonicity of step and app

theory StepMono = Step:

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

```
lemma sup_loc_some [rule_format]:
  "∀ y n. (G ⊢ b ≤l y) --> n < length y --> y!n = Ok t -->
    (∃ t. b!n = Ok t ∧ (G ⊢ (b!n) ≤o (y!n)))" (is "?P b")
proof (induct (open) ?P b)
  show "?P []" by simp

  case Cons
  show "?P (a#list)"
  proof (clarsimp simp add: list_all2_Cons1 sup_loc_def)
    fix z zs n
    assume * :
      "G ⊢ a ≤o z" "list_all2 (sup_ty_opt G) list zs"
      "n < Suc (length zs)" "(z # zs) ! n = Ok t"

    show "(∃ t. (a # list) ! n = Ok t) ∧ G ⊢ (a # list) ! n ≤o Ok t"
  proof (cases n)
    case 0
    with * show ?thesis by (simp add: sup_ty_opt_0k)
  next
    case Suc
    with Cons *
    show ?thesis by (simp add: sup_loc_def)
  qed
qed
qed
```

```
lemma all_widen_is_sup_loc:
  "∀ b. length a = length b -->
    (∀ x ∈ set (zip a b). x ∈ widen G) = (G ⊢ (map Ok a) ≤l (map Ok b))"
  (is "∀ b. length a = length b --> ?Q a b" is "?P a")
proof (induct "a")
  show "?P []" by simp

  fix l ls assume Cons: "?P ls"

  show "?P (l#ls)"
  proof (intro allI impI)
    fix b
    assume "length (l # ls) = length (b::ty list)"
    with Cons
    show "?Q (l # ls) b" by - (cases b, auto)
  qed
qed
```

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
        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
    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
      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

```

lemma app_mono:

"[|G ⊢ s <= s'; app i G rT s'|] ==> app i G rT s"

proof -

```

{ fix s1 s2
  assume G: "G ⊢ s2 <=s s1"
  assume app: "app i G rT (Some s1)"

  have "app i G rT (Some s2)"
  proof (cases (open) i)
    case Load

    from G
    have l: "length (snd s1) = length (snd s2)" by (simp add: sup_state_length)

    from G Load app
    have "G ⊢ snd s2 <=l snd s1" by (auto simp add: sup_state_def)

    with G Load app l
    show ?thesis by clarsimp (drule sup_loc_some, simp+)
  next
    case Store
    with G app
    show ?thesis
      by - (cases s2,
        auto simp add: map_eq_Cons sup_loc_Cons2 sup_loc_length sup_state_def)
  next
    case Bipush
    thus ?thesis by simp
  next
    case Aconst_null
    thus ?thesis by simp
  next
    case New
    with app
    show ?thesis by simp
  next
    case Getfield
    with app G
    show ?thesis
      by - (cases s2, clarsimp simp add: sup_state_Cons2, rule widen_trans)
  next
    case Putfield

```

```

with app
obtain vT oT ST LT b
  where s1: "s1 = (vT # oT # ST, LT)" and
    "field (G, cname) vname = Some (cname, b)"
    "is_class G cname" and
    oT: "G ⊢ oT ≤ (Class cname)" and
    vT: "G ⊢ vT ≤ b"
  by simp (elim exE conjE, rule that)
moreover
from s1 G
obtain vT' oT' ST' LT'
  where s2: "s2 = (vT' # oT' # ST', LT' )" and
    oT': "G ⊢ oT' ≤ oT" and
    vT': "G ⊢ vT' ≤ vT"
  by - (cases s2, simp add: sup_state_Cons2, elim exE conjE, simp, rule that)
moreover
from vT' vT
have "G ⊢ vT' ≤ b" by (rule widen_trans)
moreover
from oT' oT
have "G ⊢ oT' ≤ (Class cname)" by (rule widen_trans)
ultimately
show ?thesis
  by (auto simp add: Putfield)
next
case Checkcast
with app G
show ?thesis
  by - (cases s2, auto intro!: widen_RefT2 simp add: sup_state_Cons2)
next
case Return
with app G
show ?thesis
  by - (cases s2, auto simp add: sup_state_Cons2, rule widen_trans)
next
case Pop
with app G
show ?thesis
  by - (cases s2, clarsimp simp add: sup_state_Cons2)
next
case Dup
with app G
show ?thesis
  by - (cases s2, clarsimp simp add: sup_state_Cons2)
next
case Dup_x1
with app G
show ?thesis
  by - (cases s2, clarsimp simp add: sup_state_Cons2)
next
case Dup_x2

```

```

    with app G
    show ?thesis
      by - (cases s2, clarsimp simp add: sup_state_Cons2)
next
  case Swap
  with app G
  show ?thesis
    by - (cases s2, clarsimp simp add: sup_state_Cons2)
next
  case IAdd
  with app G
  show ?thesis
    by - (cases s2, auto simp add: sup_state_Cons2 PrimT_PrimT)
next
  case Goto
  with app
  show ?thesis by simp
next
  case Ifcmpeq
  with app G
  show ?thesis
    by - (cases s2, auto simp add: sup_state_Cons2 PrimT_PrimT widen_RefT2)
next
  case Invoke

  with app
  obtain apTs X ST LT mD' rT' b' where
    s1: "s1 = (rev apTs @ X # ST, LT)" and
    l: "length apTs = length list" and
    C: "G ⊢ X ≤ Class cname" and
    w: "∀x ∈ set (zip apTs list). x ∈ widen G" and
    m: "method (G, cname) (mname, list) = Some (mD', rT', b')"
    by (simp, elim exE conjE) (rule that)

  obtain apTs' X' ST' LT' where
    s2: "s2 = (rev apTs' @ X' # ST', LT')" and
    l': "length apTs' = length list"
  proof -
    from l s1 G
    have "length list < length (fst s2)"
      by (simp add: sup_state_length)
    hence "∃a b c. (fst s2) = rev a @ b # c ∧ length a = length list"
      by (rule rev_append_cons [rule_format])
    thus ?thesis
      by - (cases s2, elim exE conjE, simp, rule that)
  qed

  from l l'
  have "length (rev apTs') = length (rev apTs)" by simp

  from this s1 s2 G
  obtain

```

```

    G': "G ⊢ (apTs', LT') <=s (apTs, LT)" and
    X : "G ⊢ X' ≼ X" and "G ⊢ (ST', LT') <=s (ST, LT)"
  by (simp add: sup_state_rev_fst sup_state_append_fst sup_state_Cons1)

  with C
  have C': "G ⊢ X' ≼ Class cname"
    by - (rule widen_trans, auto)

  from G'
  have "G ⊢ map Ok apTs' <=l map Ok apTs"
    by (simp add: sup_state_def)
  also
  from l w
  have "G ⊢ map Ok apTs <=l map Ok list"
    by (simp add: all_widen_is_sup_loc)
  finally
  have "G ⊢ map Ok apTs' <=l map Ok list" .

  with l'
  have w': "∀ x ∈ set (zip apTs' list). x ∈ widen G"
    by (simp add: all_widen_is_sup_loc)

  from Invoke s2 l' w' C' m
  show ?thesis
    by simp blast
  qed
} note mono_Some = this

assume "G ⊢ s <= s'" "app i G rT s'"

thus ?thesis
  by - (cases s, cases s', auto simp add: mono_Some)
qed

lemmas [simp del] = split_paired_Ex
lemmas [simp] = step_def

lemma step_mono_Some:
  "[| succs i pc ≠ []; app i G rT (Some s2); G ⊢ s1 <=s s2 |] ==>
   G ⊢ the (step i G (Some s1)) <=s the (step i G (Some s2))"
proof (cases s1, cases s2)
  fix a1 b1 a2 b2
  assume s: "s1 = (a1, b1)" "s2 = (a2, b2)"
  assume succs: "succs i pc ≠ []"
  assume app2: "app i G rT (Some s2)"
  assume G: "G ⊢ s1 <=s s2"

  hence "G ⊢ Some s1 <= s2"
    by simp
  from this app2
  have app1: "app i G rT (Some s1)" by (rule app_mono)

```

```

have "step i G (Some s1) ≠ None ∧ step i G (Some s2) ≠ None"
  by simp
then
obtain a1' b1' a2' b2'
  where step: "step i G (Some s1) = Some (a1',b1')"
             "step i G (Some s2) = Some (a2',b2')"
  by (auto simp del: step_def simp add: s)

have "G ⊢ (a1',b1') ≤s (a2',b2')"
proof (cases (open) i)
  case Load

  with s app1
  obtain y where
    y: "nat < length b1" "b1 ! nat = 0k y" by clarsimp

  from Load s app2
  obtain y' where
    y': "nat < length b2" "b2 ! nat = 0k y'" by clarsimp

  from G s
  have "G ⊢ b1 ≤l b2" by (simp add: sup_state_def)

  with y y'
  have "G ⊢ y ≤ y'"
    by - (drule sup_loc_some, simp+)

  with Load G y y' s step app1 app2
  show ?thesis by (clarsimp simp add: sup_state_def)
next
  case Store
  with G s step app1 app2
  show ?thesis
    by (clarsimp simp add: sup_state_def sup_loc_update)
next
  case Bipush
  with G s step app1 app2
  show ?thesis
    by (clarsimp simp add: sup_state_Cons1)
next
  case New
  with G s step app1 app2
  show ?thesis
    by (clarsimp simp add: sup_state_Cons1)
next
  case Aconst_null
  with G s step app1 app2
  show ?thesis
    by (clarsimp simp add: sup_state_Cons1)
next
  case Getfield
  with G s step app1 app2

```

```

    show ?thesis
    by (clarsimp simp add: sup_state_Cons1)
next
  case Putfield
  with G s step app1 app2
  show ?thesis
    by (clarsimp simp add: sup_state_Cons1)
next
  case Checkcast
  with G s step app1 app2
  show ?thesis
    by (clarsimp simp add: sup_state_Cons1)
next
  case Invoke

  with s app1
  obtain a X ST where
    s1: "s1 = (a @ X # ST, b1)" and
    l: "length a = length list"
    by (simp, elim exE conjE, simp)

  from Invoke s app2
  obtain a' X' ST' where
    s2: "s2 = (a' @ X' # ST', b2)" and
    l': "length a' = length list"
    by (simp, elim exE conjE, simp)

  from l l'
  have lr: "length a = length a'" by simp

  from lr G s s1 s2
  have "G ⊢ (ST, b1) <=s (ST', b2)"
    by (clarsimp simp add: sup_state_append_fst sup_state_Cons1)

  moreover

  from Invoke G s step app1 app2
  have "b1 = b1' ∧ b2 = b2'" by simp

  ultimately

  have "G ⊢ (ST, b1') <=s (ST', b2')" by simp

  with Invoke G s step app1 app2 s1 s2 l l'
  show ?thesis
    by (clarsimp simp add: sup_state_def)
next
  case Return
  with succs have "False" by simp
  thus ?thesis by blast
next
  case Pop

```

```

    with G s step app1 app2
    show ?thesis
    by (clarsimp simp add: sup_state_Cons1)
next
  case Dup
  with G s step app1 app2
  show ?thesis
  by (clarsimp simp add: sup_state_Cons1)
next
  case Dup_x1
  with G s step app1 app2
  show ?thesis
  by (clarsimp simp add: sup_state_Cons1)
next
  case Dup_x2
  with G s step app1 app2
  show ?thesis
  by (clarsimp simp add: sup_state_Cons1)
next
  case Swap
  with G s step app1 app2
  show ?thesis
  by (clarsimp simp add: sup_state_Cons1)
next
  case IAdd
  with G s step app1 app2
  show ?thesis
  by (clarsimp simp add: sup_state_Cons1)
next
  case Goto
  with G s step app1 app2
  show ?thesis by simp
next
  case Ifcmpeq
  with G s step app1 app2
  show ?thesis
  by (clarsimp simp add: sup_state_Cons1)
qed

with step
show ?thesis by auto
qed

lemma step_mono:
  "[| succs i pc ≠ []; app i G rT s2; G ⊢ s1 <= s2 |] ==>
   G ⊢ step i G s1 <= step i G s2"
  by (cases s1, cases s2, auto dest: step_mono_Some)

lemmas [simp del] = step_def

end

```


27 Completeness of the LBV

theory LBVComplete = BVSpec + LBVSpec + StepMono:

constdefs

```
contains_targets :: "[instr list, certificate, method_type, p_count] => bool"
"contains_targets ins cert phi pc ==
  ∀pc'. pc' ∈ set (succs (ins!pc) pc).
  pc' ≠ pc+1 ∧ pc' < length ins --> cert!pc' = phi!pc'"
```

```
fits :: "[instr list, certificate, method_type] => bool"
"fits ins cert phi == ∀pc. pc < length ins -->
  contains_targets ins cert phi pc ∧
  (cert!pc = None ∨ cert!pc = phi!pc)"
```

```
is_target :: "[instr list, p_count] => bool"
"is_target ins pc ==
  ∃pc'. pc ≠ pc'+1 ∧ pc' < length ins ∧ pc ∈ set (succs (ins!pc') pc')"
```

constdefs

```
make_cert :: "[instr list, method_type] => certificate"
"make_cert ins phi ==
  map (λpc. if is_target ins pc then phi!pc else None) [0..length ins]"

make_Cert :: "[jvm_prog, prog_type] => prog_certificate"
"make_Cert G Phi == λ C sig.
  let (C,x,y,mdecls) = SOME (C1,x,y,mdecls). (C1,x,y,mdecls) ∈ set G ∧ C1 = C;
  (sig,rT,maxl,b) = SOME (sg,rT,maxl,b). (sg,rT,maxl,b) ∈ set mdecls ∧ sg = sig
  in make_cert b (Phi C sig)"
```

lemmas [simp del] = split_paired_Ex

lemma make_cert_target:

```
"[| pc < length ins; is_target ins pc |] ==> make_cert ins phi ! pc = phi!pc"
by (simp add: make_cert_def)
```

lemma make_cert_approx:

```
"[| pc < length ins; make_cert ins phi ! pc ≠ phi ! pc |] ==>
  make_cert ins phi ! pc = None"
by (auto simp add: make_cert_def)
```

lemma make_cert_contains_targets:

```
"pc < length ins ==> contains_targets ins (make_cert ins phi) phi pc"
```

proof (unfold contains_targets_def, intro strip, elim conjE)

```
fix pc'
assume "pc < length ins"
  "pc' ∈ set (succs (ins ! pc) pc)"
  "pc' ≠ pc+1" and
pc': "pc' < length ins"
```

```

hence "is_target ins pc'"
  by (auto simp add: is_target_def)

with pc'
show "make_cert ins phi ! pc' = phi ! pc'"
  by (auto intro: make_cert_target)
qed

theorem fits_make_cert:
  "fits ins (make_cert ins phi) phi"
  by (auto dest: make_cert_approx simp add: fits_def make_cert_contains_targets)

lemma fitsD:
  "[| fits ins cert phi; pc' ∈ set (succs (ins!pc) pc);
    pc' ≠ Suc pc; pc < length ins; pc' < length ins |]
  ==> cert!pc' = phi!pc'"
  by (clarsimp simp add: fits_def contains_targets_def)

lemma fitsD2:
  "[| fits ins cert phi; pc < length ins; cert!pc = Some s |]
  ==> cert!pc = phi!pc"
  by (auto simp add: fits_def)

lemma wtl_inst_mono:
  "[| wtl_inst i G rT s1 cert mpc pc = Ok s1'; fits ins cert phi;
    pc < length ins; G ⊢ s2 <=' s1; i = ins!pc |] ==>
  ∃ s2'. wtl_inst (ins!pc) G rT s2 cert mpc pc = Ok s2' ∧ (G ⊢ s2' <=' s1')]"
proof -
  assume pc: "pc < length ins" "i = ins!pc"
  assume wtl: "wtl_inst i G rT s1 cert mpc pc = Ok s1'"
  assume fits: "fits ins cert phi"
  assume G: "G ⊢ s2 <=' s1"

  let "?step s" = "step i G s"

  from wtl G
  have app: "app i G rT s2" by (auto simp add: wtl_inst_Ok app_mono)

  from wtl G
  have step: "succs i pc ≠ [] ==> G ⊢ ?step s2 <=' ?step s1"
    by - (drule step_mono, auto simp add: wtl_inst_Ok)

  {
    fix pc'
    assume 0: "pc' ∈ set (succs i pc)" "pc' ≠ pc+1"
    hence "succs i pc ≠ []" by auto
    hence "G ⊢ ?step s2 <=' ?step s1" by (rule step)
    also
    from wtl 0
  }

```

```

    have "G ⊢ ?step s1 ≤' cert!pc'"
      by (auto simp add: wtl_inst_0k)
    finally
      have "G ⊢ ?step s2 ≤' cert!pc'" .
  } note cert = this

have "∃ s2'. wtl_inst i G rT s2 cert mpc pc = 0k s2' ∧ G ⊢ s2' ≤' s1'"
proof (cases "pc+1 ∈ set (succs i pc)")
  case True
  with wtl
  have s1': "s1' = ?step s1" by (simp add: wtl_inst_0k)

  have "wtl_inst i G rT s2 cert mpc pc = 0k (?step s2) ∧ G ⊢ ?step s2 ≤' s1'"
    (is "?wtl ∧ ?G")
  proof
    from True s1'
    show ?G by (auto intro: step)

    from True app wtl
    show ?wtl
      by (clarsimp intro!: cert simp add: wtl_inst_0k)
  qed
  thus ?thesis by blast
next
  case False
  with wtl
  have "s1' = cert ! Suc pc" by (simp add: wtl_inst_0k)

  with False app wtl
  have "wtl_inst i G rT s2 cert mpc pc = 0k s1' ∧ G ⊢ s1' ≤' s1'"
    by (clarsimp intro!: cert simp add: wtl_inst_0k)

  thus ?thesis by blast
qed

with pc show ?thesis by simp
qed

lemma wtl_cert_mono:
  "[| wtl_cert i G rT s1 cert mpc pc = 0k s1'; fits ins cert phi;
    pc < length ins; G ⊢ s2 ≤' s1; i = ins!pc |] ==>
  ∃ s2'. wtl_cert (ins!pc) G rT s2 cert mpc pc = 0k s2' ∧ (G ⊢ s2' ≤' s1')]"
proof -
  assume wtl: "wtl_cert i G rT s1 cert mpc pc = 0k s1'" and
    fits: "fits ins cert phi" "pc < length ins"
    "G ⊢ s2 ≤' s1" "i = ins!pc"

  show ?thesis
proof (cases (open) "cert!pc")
  case None
  with wtl fits

```

```

show ?thesis
  by - (rule wtl_inst_mono [elim_format], (simp add: wtl_cert_def)+)
next
case Some
with wtl fits

have G: "G ⊢ s2 <=' (Some a)"
  by - (rule sup_state_opt_trans, auto simp add: wtl_cert_def split: if_splits)

from Some wtl
have "wtl_inst i G rT (Some a) cert mpc pc = Ok s1'"
  by (simp add: wtl_cert_def split: if_splits)

with G fits
have "∃ s2'. wtl_inst (ins!pc) G rT (Some a) cert mpc pc = Ok s2' ∧
      (G ⊢ s2' <=' s1')"
  by - (rule wtl_inst_mono, (simp add: wtl_cert_def)+)

with Some G
show ?thesis by (simp add: wtl_cert_def)
qed
qed

```

```

lemma wt_instr_imp_wtl_inst:
  "[| wt_instr (ins!pc) G rT phi max_pc pc; fits ins cert phi;
    pc < length ins; length ins = max_pc |] ==>
  wtl_inst (ins!pc) G rT (phi!pc) cert max_pc pc ≠ Err"
proof -
  assume wt: "wt_instr (ins!pc) G rT phi max_pc pc"
  assume fits: "fits ins cert phi"
  assume pc: "pc < length ins" "length ins = max_pc"

  from wt
  have app: "app (ins!pc) G rT (phi!pc)" by (simp add: wt_instr_def)

  from wt pc
  have pc': "!!pc'. pc' ∈ set (succs (ins!pc) pc) ==> pc' < length ins"
    by (simp add: wt_instr_def)

  let ?s' = "step (ins!pc) G (phi!pc)"

  from wt fits pc
  have cert: "!!pc'. [|pc' ∈ set (succs (ins!pc) pc); pc' < max_pc; pc' ≠ pc+1|]
    ==> G ⊢ ?s' <=' cert!pc'"
    by (auto dest: fitsD simp add: wt_instr_def)

  from app pc cert pc'
  show ?thesis
    by (auto simp add: wtl_inst_Ok)
qed

```

```

lemma wt_less_wtl:
  "[| wt_instr (ins!pc) G rT phi max_pc pc;
    wtl_inst (ins!pc) G rT (phi!pc) cert max_pc pc = Ok s;
    fits ins cert phi; Suc pc < length ins; length ins = max_pc |] ==>
  G ⊢ s <= ' phi ! Suc pc"
proof -
  assume wt: "wt_instr (ins!pc) G rT phi max_pc pc"
  assume wtl: "wtl_inst (ins!pc) G rT (phi!pc) cert max_pc pc = Ok s"
  assume fits: "fits ins cert phi"
  assume pc: "Suc pc < length ins" "length ins = max_pc"

  { assume suc: "Suc pc ∈ set (sucss (ins ! pc) pc)"
    with wtl have "s = step (ins!pc) G (phi!pc)"
      by (simp add: wtl_inst_Ok)
    also from suc wt have "G ⊢ ... <= ' phi!Suc pc"
      by (simp add: wt_instr_def)
    finally have ?thesis .
  }

  moreover
  { assume "Suc pc ∉ set (sucss (ins ! pc) pc)"

    with wtl
    have "s = cert!Suc pc"
      by (simp add: wtl_inst_Ok)
    with fits pc
    have ?thesis
      by - (cases "cert!Suc pc", simp, drule fitsD2, assumption+, simp)
  }

  ultimately
  show ?thesis by blast
qed

```

```

lemma wt_instr_imp_wtl_cert:
  "[| wt_instr (ins!pc) G rT phi max_pc pc; fits ins cert phi;
    pc < length ins; length ins = max_pc |] ==>
  wtl_cert (ins!pc) G rT (phi!pc) cert max_pc pc ≠ Err"
proof -
  assume "wt_instr (ins!pc) G rT phi max_pc pc" and
  fits: "fits ins cert phi" and
  pc: "pc < length ins" and
  "length ins = max_pc"

  hence wtl: "wtl_inst (ins!pc) G rT (phi!pc) cert max_pc pc ≠ Err"
    by (rule wt_instr_imp_wtl_inst)

  hence "cert!pc = None ==> ?thesis"
    by (simp add: wtl_cert_def)

  moreover

```

```

{ fix s
  assume c: "cert!pc = Some s"
  with fits pc
  have "cert!pc = phi!pc"
    by (rule fitsD2)
  from this c wtl
  have ?thesis
    by (clarsimp simp add: wtl_cert_def)
}

ultimately
show ?thesis by blast
qed

lemma wt_less_wtl_cert:
  "[| wt_instr (ins!pc) G rT phi max_pc pc;
    wtl_cert (ins!pc) G rT (phi!pc) cert max_pc pc = Ok s;
    fits ins cert phi; Suc pc < length ins; length ins = max_pc |] ==>
  G ⊢ s <= ' phi ! Suc pc"
proof -
  assume wtl: "wtl_cert (ins!pc) G rT (phi!pc) cert max_pc pc = Ok s"

  assume wti: "wt_instr (ins!pc) G rT phi max_pc pc"
    "fits ins cert phi"
    "Suc pc < length ins" "length ins = max_pc"

  { assume "cert!pc = None"
    with wtl
    have "wtl_inst (ins!pc) G rT (phi!pc) cert max_pc pc = Ok s"
      by (simp add: wtl_cert_def)
    with wti
    have ?thesis
      by - (rule wt_less_wtl)
  }
  moreover
  { fix t
    assume c: "cert!pc = Some t"
    with wti
    have "cert!pc = phi!pc"
      by - (rule fitsD2, simp+)
    with c wtl
    have "wtl_inst (ins!pc) G rT (phi!pc) cert max_pc pc = Ok s"
      by (simp add: wtl_cert_def)
    with wti
    have ?thesis
      by - (rule wt_less_wtl)
  }

  ultimately
  show ?thesis by blast
qed

```

Main induction over the instruction list.

theorem *wt_imp_wtl_inst_list*:

```

"∀ pc. (∀ pc'. pc' < length all_ins -->
  wt_instr (all_ins ! pc') G rT phi (length all_ins) pc') -->
  fits all_ins cert phi -->
  (∃ l. pc = length l ∧ all_ins = l@ins) -->
  pc < length all_ins -->
  (∀ s. (G ⊢ s <= ' (phi!pc))) -->
  wtl_inst_list ins G rT cert (length all_ins) pc s ≠ Err)"
(is "∀ pc. ?wt --> ?fits --> ?l pc ins --> ?len pc --> ?wtl pc ins"
  is "∀ pc. ?C pc ins" is "?P ins")
proof (induct "?P" "ins")
  case Nil
  show "?P []" by simp
next
  fix i ins'
  assume Cons: "?P ins'"

  show "?P (i#ins')"
  proof (intro allI impI, elim exE conjE)
    fix pc s l
    assume wt : "∀ pc'. pc' < length all_ins -->
      wt_instr (all_ins ! pc') G rT phi (length all_ins) pc'"
    assume fits: "fits all_ins cert phi"
    assume l : "pc < length all_ins"
    assume G : "G ⊢ s <= ' phi ! pc"
    assume pc : "all_ins = l@i#ins'" "pc = length l"
    hence i : "all_ins ! pc = i"
      by (simp add: nth_append)

    from l wt
    have wti: "wt_instr (all_ins!pc) G rT phi (length all_ins) pc" by blast

    with fits l
    have c: "wtl_cert (all_ins!pc) G rT (phi!pc) cert (length all_ins) pc ≠ Err"
      by - (drule wt_instr_imp_wtl_cert, auto)

    from Cons
    have "?C (Suc pc) ins'" by blast
    with wt fits pc
    have IH: "?len (Suc pc) --> ?wtl (Suc pc) ins'" by auto

    show "wtl_inst_list (i#ins') G rT cert (length all_ins) pc s ≠ Err"
    proof (cases "?len (Suc pc)")
      case False
      with pc
      have "ins' = []" by simp
      with G i c fits l
      show ?thesis by (auto dest: wtl_cert_mono)
    next
      case True
      with IH

```

```

have wtl: "?wtl (Suc pc) ins'" by blast

from c wti fits l True
obtain s'' where
  "wtl_cert (all_ins!pc) G rT (phi!pc) cert (length all_ins) pc = Ok s'"
  "G ⊢ s'' ≤' phi ! Suc pc"
  by clarsimp (drule wt_less_wtl_cert, auto)
moreover
from calculation fits G l
obtain s' where
  c': "wtl_cert (all_ins!pc) G rT s cert (length all_ins) pc = Ok s'" and
  "G ⊢ s' ≤' s'"
  by - (drule wtl_cert_mono, auto)
ultimately
have G': "G ⊢ s' ≤' phi ! Suc pc"
  by - (rule sup_state_opt_trans)

with wtl
have "wtl_inst_list ins' G rT cert (length all_ins) (Suc pc) s' ≠ Err"
  by auto

with i c'
show ?thesis by auto
qed
qed
qed

lemma fits_imp_wtl_method_complete:
  "[| wt_method G C pTs rT mxl ins phi; fits ins cert phi |] ==>
   wtl_method G C pTs rT mxl ins cert"
by (simp add: wt_method_def wtl_method_def)
  (rule wt_imp_wtl_inst_list [rule_format, elim_format], auto simp add: wt_start_def)

lemma wtl_method_complete:
  "wt_method G C pTs rT mxl ins phi ==>
   wtl_method G C pTs rT mxl ins (make_cert ins phi)"
proof -
  assume "wt_method G C pTs rT mxl ins phi"
  moreover
  have "fits ins (make_cert ins phi) phi"
    by (rule fits_make_cert)
  ultimately
  show ?thesis
    by (rule fits_imp_wtl_method_complete)
qed

theorem wtl_complete:
  "wt_jvm_prog G Phi ==> wtl_jvm_prog G (make_Cert G Phi)"
proof (unfold wt_jvm_prog_def)

```

```

assume wfprog:
  "wf_prog ( $\lambda G C (sig, rT, maxl, b). wt\_method G C (snd sig) rT maxl b (Phi C sig)) G"$ 

thus ?thesis
proof (simp add: wtl_jvm_prog_def wf_prog_def wf_cdecl_def wf_mdecl_def, auto)
  fix a aa ab b ac ba ad ae bb
  assume 1 : " $\forall (C, sc, fs, ms) \in set G.$ 
    Ball (set fs) (wf_fdecl G)  $\wedge$ 
    unique fs  $\wedge$ 
    ( $\forall (sig, rT, mb) \in set ms. wf\_mhead G sig rT \wedge$ 
      ( $\lambda (maxl, b). wt\_method G C (snd sig) rT maxl b (Phi C sig)) mb$ )  $\wedge$ 
    unique ms  $\wedge$ 
    (case sc of None  $\Rightarrow C = Object$ 
      | Some D  $\Rightarrow$ 
        is_class G D  $\wedge$ 
        (D, C)  $\notin (subcls1 G)^*$   $\wedge$ 
        ( $\forall (sig, rT, b) \in set ms.$ 
           $\forall D' rT' b'. method (G, D) sig = Some (D', rT', b') \rightarrow G \vdash rT \preceq rT')$ )"
    "(a, aa, ab, b)  $\in set G$ "

  assume uG : "unique G"
  assume b : " $((ac, ba), ad, ae, bb) \in set b$ "

  from 1
  show "wtl_method G a ba ad ae bb (make_Cert G Phi a (ac, ba))"
  proof (rule bspec [elim_format], clarsimp)
    assume ub : "unique b"
    assume m: " $\forall (sig, rT, mb) \in set b. wf\_mhead G sig rT \wedge$ 
      ( $\lambda (maxl, b). wt\_method G a (snd sig) rT maxl b (Phi a sig)) mb$ "
    from m b
    show ?thesis
    proof (rule bspec [elim_format], clarsimp)
      assume "wt_method G a ba ad ae bb (Phi a (ac, ba))"
      with wfprog uG ub b 1
      show ?thesis
      by - (rule wtl_method_complete [elim_format], assumption+,
        simp add: make_Cert_def unique_epsilon)
    qed
  qed
qed
qed
qed

lemmas [simp] = split_paired_Ex

end

```

28 Theorem Digest

theory Digest = JTypeSafe + Example + BVSpecTypeSafe + LBVCorrect + LBVComplete:

Theory BVSpec

lemma wt_jvm_progD:

wt_jvm_prog G phi $\implies \exists wt. wf_prog wt G$

lemma wt_jvm_prog_impl_wt_instr:

$\llbracket wt_jvm_prog G phi; method (G, C) sig = Some (C, rT, maxl, ins);$
 $pc < length ins \rrbracket$
 $\implies wt_instr (ins ! pc) G rT (phi C sig) (length ins) pc$

lemma wt_jvm_prog_impl_wt_start:

$\llbracket wt_jvm_prog G phi; method (G, C) sig = Some (C, rT, maxl, ins) \rrbracket$
 $\implies 0 < length ins \wedge wt_start G C (snd sig) maxl (phi C sig)$

Theory BVSpecTypeSafe

lemma wt_jvm_prog_impl_wt_instr_cor:

$\llbracket wt_jvm_prog G phi; method (G, C) sig = Some (C, rT, maxl, ins);$
 $G, phi \vdash JVM Norm (hp, (stk, loc, C, sig, pc) \# frs) \checkmark \rrbracket$
 $\implies wt_instr (ins ! pc) G rT (phi C sig) (length ins) pc$

lemma Load_correct:

$\llbracket wf_prog wt G; method (G, C) sig = Some (C, rT, maxl, ins);$
 $ins ! pc = Load idx;$
 $wt_instr (ins ! pc) G rT (phi C sig) (length ins) pc;$
 $Some state' = JVMExec.exec (G, Norm (hp, (stk, loc, C, sig, pc) \# frs));$
 $G, phi \vdash JVM Norm (hp, (stk, loc, C, sig, pc) \# frs) \checkmark \rrbracket$
 $\implies G, phi \vdash JVM state' \checkmark$

lemma Store_correct:

$\llbracket wf_prog wt G; method (G, C) sig = Some (C, rT, maxl, ins);$
 $ins ! pc = Store idx;$
 $wt_instr (ins ! pc) G rT (phi C sig) (length ins) pc;$
 $Some state' = JVMExec.exec (G, Norm (hp, (stk, loc, C, sig, pc) \# frs));$
 $G, phi \vdash JVM Norm (hp, (stk, loc, C, sig, pc) \# frs) \checkmark \rrbracket$
 $\implies G, phi \vdash JVM state' \checkmark$

lemma conf_Intg_Integer:

$G, h \vdash Intg i :: \preceq PrimT Integer$

lemma Bipush_correct:

$\llbracket wf_prog wt G; method (G, C) sig = Some (C, rT, maxl, ins);$
 $ins ! pc = Bipush i;$
 $wt_instr (ins ! pc) G rT (phi C sig) (length ins) pc;$
 $Some state' = JVMExec.exec (G, Norm (hp, (stk, loc, C, sig, pc) \# frs));$
 $G, phi \vdash JVM Norm (hp, (stk, loc, C, sig, pc) \# frs) \checkmark \rrbracket$
 $\implies G, phi \vdash JVM state' \checkmark$

lemma NT_subtype_conv:

$G \vdash NT \preceq T = (T = NT \vee (\exists C. T = Class C))$

lemma Aconst_null_correct:

```

[[wf_prog wt G; method (G, C) sig = Some (C, rT, maxl, ins);
  ins ! pc = Aconst_null;
  wt_instr (ins ! pc) G rT (phi C sig) (length ins) pc;
  Some state' = JVMEExec.exec (G, Norm (hp, (stk, loc, C, sig, pc) # frs));
  G, phi ⊢ JVM Norm (hp, (stk, loc, C, sig, pc) # frs) √]]
⇒ G, phi ⊢ JVM state' √

```

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

```

lemma Checkcast_correct:

```

[[wf_prog wt G; method (G, C) sig = Some (C, rT, maxl, ins);
  ins ! pc = Checkcast D;
  wt_instr (ins ! pc) G rT (phi C sig) (length ins) pc;
  Some state' = JVMEExec.exec (G, Norm (hp, (stk, loc, C, sig, pc) # frs));
  G, phi ⊢ JVM Norm (hp, (stk, loc, C, sig, pc) # frs) √]]
⇒ G, phi ⊢ JVM state' √

```

lemma Getfield_correct:

```

[[wf_prog wt G; method (G, C) sig = Some (C, rT, maxl, ins);
  ins ! pc = Getfield F D;
  wt_instr (ins ! pc) G rT (phi C sig) (length ins) pc;
  Some state' = JVMEExec.exec (G, Norm (hp, (stk, loc, C, sig, pc) # frs));
  G, phi ⊢ JVM Norm (hp, (stk, loc, C, sig, pc) # frs) √]]
⇒ G, phi ⊢ JVM state' √

```

lemma Putfield_correct:

```

[[wf_prog wt G; method (G, C) sig = Some (C, rT, maxl, ins);
  ins ! pc = Putfield F D;
  wt_instr (ins ! pc) G rT (phi C sig) (length ins) pc;
  Some state' = JVMEExec.exec (G, Norm (hp, (stk, loc, C, sig, pc) # frs));
  G, phi ⊢ JVM Norm (hp, (stk, loc, C, sig, pc) # frs) √]]
⇒ G, phi ⊢ JVM state' √

```

lemma collapse_paired_All:

$(\forall x y. P(x, y)) = (\forall z. P z)$

lemma New_correct:

```

[[wf_prog wt G; method (G, C) sig = Some (C, rT, maxl, ins);
  ins ! pc = New cl_idx;
  wt_instr (ins ! pc) G rT (phi C sig) (length ins) pc;
  Some state' = JVMEExec.exec (G, Norm (hp, (stk, loc, C, sig, pc) # frs));
  G, phi ⊢ JVM Norm (hp, (stk, loc, C, sig, pc) # frs) √]]
⇒ G, phi ⊢ JVM state' √

```

lemma Invoke_correct:

```

[[wt_jvm_prog G phi; method (G, C) sig = Some (C, rT, maxl, ins);
  ins ! pc = Invoke C' mn pTs;
  wt_instr (ins ! pc) G rT (phi C sig) (length ins) pc;
  Some state' = JVMEExec.exec (G, Norm (hp, (stk, loc, C, sig, pc) # frs));
  G, phi ⊢ JVM Norm (hp, (stk, loc, C, sig, pc) # frs) √]]
⇒ G, phi ⊢ JVM state' √

```

lemma Return_correct:

```

[[wt_jvm_prog G phi; method (G, C) sig = Some (C, rT, maxl, ins);
  ins ! pc = Return; wt_instr (ins ! pc) G rT (phi C sig) (length ins) pc;
  Some state' = JVMEExec.exec (G, Norm (hp, (stk, loc, C, sig, pc) # frs));
  G, phi ⊢ JVM Norm (hp, (stk, loc, C, sig, pc) # frs) √]]
⇒ G, phi ⊢ JVM state' √

```

lemma Goto_correct:

```

[[wf_prog wt G; method (G, C) sig = Some (C, rT, maxl, ins);
  ins ! pc = Goto branch;
  wt_instr (ins ! pc) G rT (phi C sig) (length ins) pc;
  Some state' = JVMEExec.exec (G, Norm (hp, (stk, loc, C, sig, pc) # frs));
  G, phi ⊢ JVM Norm (hp, (stk, loc, C, sig, pc) # frs) √]]
⇒ G, phi ⊢ JVM state' √

```

lemma Ifcmpeq_correct:

```

[[wf_prog wt G; method (G, C) sig = Some (C, rT, maxl, ins);
  ins ! pc = Ifcmpeq branch;
  wt_instr (ins ! pc) G rT (phi C sig) (length ins) pc;
  Some state' = JVMEExec.exec (G, Norm (hp, (stk, loc, C, sig, pc) # frs));
  G, phi ⊢ JVM Norm (hp, (stk, loc, C, sig, pc) # frs) √]]
⇒ G, phi ⊢ JVM state' √

```

lemma Pop_correct:

```

[[wf_prog wt G; method (G, C) sig = Some (C, rT, maxl, ins); ins ! pc = Pop;
  wt_instr (ins ! pc) G rT (phi C sig) (length ins) pc;
  Some state' = JVMEExec.exec (G, Norm (hp, (stk, loc, C, sig, pc) # frs));
  G, phi ⊢ JVM Norm (hp, (stk, loc, C, sig, pc) # frs) √]]
⇒ G, phi ⊢ JVM state' √

```

lemma Dup_correct:

```

[[wf_prog wt G; method (G, C) sig = Some (C, rT, maxl, ins); ins ! pc = Dup;
  wt_instr (ins ! pc) G rT (phi C sig) (length ins) pc;
  Some state' = JVMEExec.exec (G, Norm (hp, (stk, loc, C, sig, pc) # frs));
  G, phi ⊢ JVM Norm (hp, (stk, loc, C, sig, pc) # frs) √]]
⇒ G, phi ⊢ JVM state' √

```

lemma Dup_x1_correct:

```

[[wf_prog wt G; method (G, C) sig = Some (C, rT, maxl, ins);
  ins ! pc = Dup_x1; wt_instr (ins ! pc) G rT (phi C sig) (length ins) pc;
  Some state' = JVMEExec.exec (G, Norm (hp, (stk, loc, C, sig, pc) # frs));
  G, phi ⊢ JVM Norm (hp, (stk, loc, C, sig, pc) # frs) √]]
⇒ G, phi ⊢ JVM state' √

```

lemma Dup_x2_correct:

```

[[wf_prog wt G; method (G, C) sig = Some (C, rT, maxl, ins);
  ins ! pc = Dup_x2; wt_instr (ins ! pc) G rT (phi C sig) (length ins) pc;
  Some state' = JVMEExec.exec (G, Norm (hp, (stk, loc, C, sig, pc) # frs));
  G, phi ⊢ JVM Norm (hp, (stk, loc, C, sig, pc) # frs) √]]
⇒ G, phi ⊢ JVM state' √

```

lemma Swap_correct:

```

[[wf_prog wt G; method (G, C) sig = Some (C, rT, maxl, ins); ins ! pc = Swap;
  wt_instr (ins ! pc) G rT (phi C sig) (length ins) pc;
  Some state' = JVMEExec.exec (G, Norm (hp, (stk, loc, C, sig, pc) # frs));

```

$G, \text{phi} \vdash \text{JVM Norm (hp, (stk, loc, C, sig, pc) \# frs)} \checkmark$
 $\Rightarrow G, \text{phi} \vdash \text{JVM state}' \checkmark$

lemma *IAdd_correct*:

$\llbracket \text{wf_prog wt } G; \text{ method (G, C) sig = Some (C, rT, maxl, ins); ins ! pc = IAdd;} \\ \text{wt_instr (ins ! pc) G rT (phi C sig) (length ins) pc;} \\ \text{Some state}' = \text{JVMEExec.exec (G, Norm (hp, (stk, loc, C, sig, pc) \# frs));} \\ G, \text{phi} \vdash \text{JVM Norm (hp, (stk, loc, C, sig, pc) \# frs)} \checkmark \rrbracket$
 $\Rightarrow G, \text{phi} \vdash \text{JVM state}' \checkmark$

lemma *instr_correct*:

$\llbracket \text{wt_jvm_prog G phi; method (G, C) sig = Some (C, rT, maxl, ins);} \\ \text{Some state}' = \text{JVMEExec.exec (G, Norm (hp, (stk, loc, C, sig, pc) \# frs));} \\ G, \text{phi} \vdash \text{JVM Norm (hp, (stk, loc, C, sig, pc) \# frs)} \checkmark \rrbracket$
 $\Rightarrow G, \text{phi} \vdash \text{JVM state}' \checkmark$

lemma *correct_state_impl_Some_method*:

$G, \text{phi} \vdash \text{JVM Norm (hp, (stk, loc, C, sig, pc) \# frs)} \checkmark \Rightarrow$
 $\exists \text{meth. method (G, C) sig = Some (C, meth)}$

lemma *BV_correct_1*:

$\llbracket \text{wt_jvm_prog G phi; G, phi} \vdash \text{JVM state} \checkmark; \\ \text{JVMEExec.exec (G, state)} = \text{Some state}' \rrbracket$
 $\Rightarrow G, \text{phi} \vdash \text{JVM state}' \checkmark$

lemma *L0*:

$\llbracket \text{xp = None; frs} \neq [] \rrbracket \Rightarrow \exists \text{state}'. \text{JVMEExec.exec (G, xp, hp, frs)} = \text{Some state}'$

lemma *L1*:

$\llbracket \text{wt_jvm_prog G phi; G, phi} \vdash \text{JVM (xp, hp, frs)} \checkmark; \text{xp = None; frs} \neq [] \rrbracket$
 $\Rightarrow \exists \text{state}'. \text{JVMEExec.exec (G, xp, hp, frs)} = \text{Some state}' \wedge G, \text{phi} \vdash \text{JVM state}' \checkmark$

theorem *BV_correct*:

$\llbracket \text{wt_jvm_prog G phi; G} \vdash s \text{ -jvm-} \rightarrow t; G, \text{phi} \vdash \text{JVM } s \checkmark \rrbracket \Rightarrow G, \text{phi} \vdash \text{JVM } t \checkmark$

theorem *BV_correct_initial*:

$\llbracket \text{wt_jvm_prog G phi; G} \vdash s_0 \text{ -jvm-} \rightarrow \text{Norm (hp, (stk, loc, C, sig, pc) \# frs);} \\ G, \text{phi} \vdash \text{JVM } s_0 \checkmark \rrbracket$
 $\Rightarrow \text{approx_stk G hp stk (fst (the (phi C sig ! pc)))} \wedge$
 $\text{approx_loc G hp loc (snd (the (phi C sig ! pc)))}$

Theory Conform

theorem *conf_VoidI*:

$G, s \vdash \text{Unit} :: \preceq \text{PrimT Void}$

theorem *conf_BooleanI*:

$G, s \vdash \text{Bool } b :: \preceq \text{PrimT Boolean}$

theorem *conf_IntegerI*:

$G, s \vdash \text{Intg } i :: \preceq \text{PrimT Integer}$

theorem *defval_conf*:

$\text{is_type G T} \Rightarrow G, h \vdash \text{default_val T} :: \preceq T$

theorem *conf_widen*:

$\llbracket \text{wf_prog wf_mb G; G, h} \vdash x :: \preceq T; G \vdash T \preceq T' \rrbracket \Rightarrow G, h \vdash x :: \preceq T'$

theorem *conf_hext*:

$\llbracket h \leq / h'; G, h \vdash xa :: \preceq x \rrbracket \implies G, h' \vdash xa :: \preceq x$

theorem *conf_RefTD*:

$G, h \vdash a' :: \preceq \text{RefT } T \implies$

$a' = \text{Null} \vee$

$(\exists a \text{ obj } T'. \quad$

$a' = \text{Addr } a \wedge h \ a = \text{Some } \text{obj} \wedge \text{obj_ty } \text{obj} = T' \wedge G \vdash T' \preceq \text{RefT } T)$

theorem *non_np_objD'*:

$\llbracket a' \neq \text{Null}; \text{wf_prog } \text{wf_mb } G; G, h \vdash a' :: \preceq \text{RefT } t; \quad$

$\forall C. t = \text{ClassT } C \longrightarrow C \neq \text{Object} \rrbracket$

$\implies \exists a \ C \ fs. a' = \text{Addr } a \wedge h \ a = \text{Some } (C, fs) \wedge G \vdash \text{Class } C \preceq \text{RefT } t$

theorem *conf_list_gext_widen*:

$\llbracket \text{wf_prog } \text{wf_mb } G; \text{list_all2 } (\text{conf } G \ h) \ \text{vs } Ts; \quad$

$\text{list_all2 } (\lambda T \ T'. G \vdash T \preceq T') \ Ts \ Ts' \rrbracket$

$\implies \text{list_all2 } (\text{conf } G \ h) \ \text{vs } Ts'$

theorem *lconf_upd*:

$\llbracket G, h \vdash l \ [:: \preceq] \ 1T; G, h \vdash v :: \preceq T; 1T \ va = \text{Some } T \rrbracket \implies G, h \vdash l(va \mapsto v) \ [:: \preceq] \ 1T$

theorem *lconf_init_vars_lemma*:

$\llbracket \forall x. P \ x \longrightarrow R \ (dv \ x) \ x; \forall x. \text{map_of } fs \ f = \text{Some } x \longrightarrow P \ x; \text{map_of } fs \ f = \text{Some } T \rrbracket$

$\implies \exists v. \text{map_of } (\text{map } (\lambda(f, ft). (f, dv \ ft)) \ fs) \ f = \text{Some } v \wedge R \ v \ T$

theorem *lconf_init_vars*:

$\forall n \ T. \text{map_of } fs \ n = \text{Some } T \longrightarrow \text{is_type } G \ T \implies$

$G, h \vdash \text{init_vars } fs \ [:: \preceq] \ \text{map_of } fs$

theorem *lconf_ext_list*:

$\llbracket G, h \vdash l \ [:: \preceq] \ L; \text{nodups } vns; \text{length } Ts = \text{length } vns; \quad$

$\text{list_all2 } (\text{conf } G \ h) \ \text{vs } Ts \rrbracket$

$\implies G, h \vdash l(vns[\mapsto]vs) \ [:: \preceq] \ L(vns[\mapsto]Ts)$

theorem *hconfD*:

$\llbracket G \vdash h \ h \ \checkmark; h \ a = \text{Some } \text{obj} \rrbracket \implies G, h \vdash \text{obj } \checkmark$

theorem *hconfI*:

$\forall a \ \text{obj}. h \ a = \text{Some } \text{obj} \longrightarrow G, h \vdash \text{obj } \checkmark \implies G \vdash h \ h \ \checkmark$

theorem *conforms_hext*:

$\llbracket (h, l) :: \preceq (G, 1T); h \leq / h'; G \vdash h \ h' \ \checkmark \rrbracket \implies (h', l) :: \preceq (G, 1T)$

theorem *conforms_upd_obj*:

$\llbracket (h, l) :: \preceq (G, 1T); G, h(a \mapsto \text{obj}) \vdash \text{obj } \checkmark; h \leq / h(a \mapsto \text{obj}) \rrbracket$

$\implies (h(a \mapsto \text{obj}), l) :: \preceq (G, 1T)$

theorem *conforms_upd_local*:

$\llbracket (h, l) :: \preceq (G, 1T); G, h \vdash v :: \preceq T; 1T \ va = \text{Some } T \rrbracket$

$\implies (h, l(va \mapsto v)) :: \preceq (G, 1T)$

Theory Convert

lemma *not_Err_eq*:

$(x \neq \text{Err}) = (\exists a. x = \text{Ok } a)$

lemma *not_Some_eq*:

$(\forall y. x \neq \text{Some } y) = (x = \text{None})$

lemma *lift_top_refl*:

$(\bigwedge x. P \ x \ x) \implies \text{lift_top } P \ x \ x$

lemma *lift_top_trans*:

$\llbracket \bigwedge x \ y \ z. \llbracket P \ x \ y; P \ y \ z \rrbracket \implies P \ x \ z; \text{lift_top } P \ x \ y; \text{lift_top } P \ y \ z \rrbracket$
 $\implies \text{lift_top } P \ x \ z$

lemma *lift_top_Err_any*:

$\text{lift_top } P \ \text{Err} \ \text{any} = (\text{any} = \text{Err})$

lemma *lift_top_Ok_Ok*:

$\text{lift_top } P \ (\text{Ok } a) \ (\text{Ok } b) = P \ a \ b$

lemma *lift_top_any_Ok*:

$\text{lift_top } P \ \text{any} \ (\text{Ok } b) = (\exists a. \text{any} = \text{Ok } a \wedge P \ a \ b)$

lemma *lift_top_Ok_any*:

$\text{lift_top } P \ (\text{Ok } a) \ \text{any} = (\text{any} = \text{Err} \vee (\exists b. \text{any} = \text{Ok } b \wedge P \ a \ b))$

lemma *lift_bottom_refl*:

$(\bigwedge x. P \ x \ x) \implies \text{lift_bottom } P \ x \ x$

lemma *lift_bottom_trans*:

$\llbracket \bigwedge x \ y \ z. \llbracket P \ x \ y; P \ y \ z \rrbracket \implies P \ x \ z; \text{lift_bottom } P \ x \ y; \text{lift_bottom } P \ y \ z \rrbracket$
 $\implies \text{lift_bottom } P \ x \ z$

lemma *lift_bottom_any_None*:

$\text{lift_bottom } P \ \text{any} \ \text{None} = (\text{any} = \text{None})$

lemma *lift_bottom_Some_Some*:

$\text{lift_bottom } P \ (\text{Some } a) \ (\text{Some } b) = P \ a \ b$

lemma *lift_bottom_any_Some*:

$\text{lift_bottom } P \ (\text{Some } a) \ \text{any} = (\exists b. \text{any} = \text{Some } b \wedge P \ a \ b)$

lemma *lift_bottom_Some_any*:

$\text{lift_bottom } P \ \text{any} \ (\text{Some } b) = (\text{any} = \text{None} \vee (\exists a. \text{any} = \text{Some } a \wedge P \ a \ b))$

theorem *sup_ty_opt_refl*:

$G \vdash t \leq_o t$

theorem *sup_loc_refl*:

$G \vdash t \leq_l t$

theorem *sup_state_refl*:

$G \vdash s \leq_s s$

theorem *sup_state_opt_refl*:

$G \vdash s \leq' s$

theorem *anyConvErr*:

$G \vdash \text{Err} \leq_o \text{any} = (\text{any} = \text{Err})$

theorem *OkanyConvOk*:

$G \vdash \text{Ok } ty' \leq_o \text{Ok } ty = G \vdash ty' \preceq ty$

theorem *sup_ty_opt_Ok*:

$G \vdash a \leq_o \text{Ok } b \implies \exists x. a = \text{Ok } x$

lemma widen_PrimT_conv1:

$\llbracket G \vdash S \preceq T; S = \text{PrimT } x \rrbracket \implies T = \text{PrimT } x$

theorem sup_PTS_eq:

$G \vdash \text{Ok } (\text{PrimT } p) \leq_o X = (X = \text{Err} \vee X = \text{Ok } (\text{PrimT } p))$

theorem sup_loc_Nil:

$G \vdash [] \leq_l XT = (XT = [])$

theorem sup_loc_Cons:

$G \vdash (Y \# YT) \leq_l XT = (\exists X XT'. XT = X \# XT' \wedge G \vdash Y \leq_o X \wedge G \vdash YT \leq_l XT')$

theorem sup_loc_Cons2:

$G \vdash YT \leq_l X \# XT = (\exists Y YT'. YT = Y \# YT' \wedge G \vdash Y \leq_o X \wedge G \vdash YT' \leq_l XT)$

theorem sup_loc_length:

$G \vdash a \leq_l b \implies \text{length } a = \text{length } b$

theorem sup_loc_nth:

$\llbracket G \vdash a \leq_l b; n < \text{length } a \rrbracket \implies G \vdash a ! n \leq_o b ! n$

theorem all_nth_sup_loc:

$\forall b. \text{length } a = \text{length } b \longrightarrow$
 $(\forall n. n < \text{length } a \longrightarrow G \vdash a ! n \leq_o b ! n) \longrightarrow G \vdash a \leq_l b$

theorem sup_loc_append:

$\text{length } a = \text{length } b \implies G \vdash (a @ x) \leq_l b @ y = (G \vdash a \leq_l b \wedge G \vdash x \leq_l y)$

theorem sup_loc_rev:

$G \vdash \text{rev } a \leq_l \text{rev } b = G \vdash a \leq_l b$

theorem sup_loc_update:

$\llbracket G \vdash a \leq_o b; n < \text{length } y; G \vdash x \leq_l y \rrbracket \implies G \vdash x[n := a] \leq_l y[n := b]$

theorem sup_state_length:

$G \vdash s2 \leq_s s1 \implies$
 $\text{length } (\text{fst } s2) = \text{length } (\text{fst } s1) \wedge \text{length } (\text{snd } s2) = \text{length } (\text{snd } s1)$

theorem sup_state_append_snd:

$\text{length } a = \text{length } b \implies$
 $G \vdash (i, a @ x) \leq_s (j, b @ y) =$
 $(G \vdash (i, a) \leq_s (j, b) \wedge G \vdash (i, x) \leq_s (j, y))$

theorem sup_state_append_fst:

$\text{length } a = \text{length } b \implies$
 $G \vdash (a @ x, i) \leq_s (b @ y, j) =$
 $(G \vdash (a, i) \leq_s (b, j) \wedge G \vdash (x, i) \leq_s (y, j))$

theorem sup_state_Cons1:

$G \vdash (x \# xt, a) \leq_s (yt, b) =$
 $(\exists y yt'. yt = y \# yt' \wedge G \vdash x \preceq y \wedge G \vdash (xt, a) \leq_s (yt', b))$

theorem sup_state_Cons2:

$G \vdash (xt, a) \leq_s (y \# yt, b) =$
 $(\exists x xt'. xt = x \# xt' \wedge G \vdash x \preceq y \wedge G \vdash (xt', a) \leq_s (yt, b))$

theorem sup_state_ignore_fst:

$G \vdash (a, x) \leq_s (b, y) \implies G \vdash (c, x) \leq_s (c, y)$

theorem *sup_state_rev_fst*:

$G \vdash (\text{rev } a, x) \leq_s (\text{rev } b, y) = G \vdash (a, x) \leq_s (b, y)$

lemma *sup_state_opt_None_any*:

$G \vdash \text{None} \leq' \text{any} = \text{True}$

lemma *sup_state_opt_any_None*:

$G \vdash \text{any} \leq' \text{None} = (\text{any} = \text{None})$

lemma *sup_state_opt_Some_Some*:

$G \vdash \text{Some } a \leq' \text{Some } b = G \vdash a \leq_s b$

lemma *sup_state_opt_any_Some*:

$G \vdash \text{Some } a \leq' \text{any} = (\exists b. \text{any} = \text{Some } b \wedge G \vdash a \leq_s b)$

lemma *sup_state_opt_Some_any*:

$G \vdash \text{any} \leq' \text{Some } b = (\text{any} = \text{None} \vee (\exists a. \text{any} = \text{Some } a \wedge G \vdash a \leq_s b))$

theorem *sup_ty_opt_trans*:

$\llbracket G \vdash a \leq_o b; G \vdash b \leq_o c \rrbracket \implies G \vdash a \leq_o c$

theorem *sup_loc_trans*:

$\llbracket G \vdash a \leq_l b; G \vdash b \leq_l c \rrbracket \implies G \vdash a \leq_l c$

theorem *sup_state_trans*:

$\llbracket G \vdash a \leq_s b; G \vdash b \leq_s c \rrbracket \implies G \vdash a \leq_s c$

theorem *sup_state_opt_trans*:

$\llbracket G \vdash a \leq' b; G \vdash b \leq' c \rrbracket \implies G \vdash a \leq' c$

Theory Correct

lemma *sup_heap_newref*:

$\text{hp } x = \text{None} \implies \text{hp} \leq_l \text{hp}(\text{newref } \text{hp} \mapsto \text{obj})$

lemma *sup_heap_update_value*:

$\text{hp } a = \text{Some } (C, \text{od}') \implies \text{hp} \leq_l \text{hp}(a \mapsto (C, \text{od}))$

lemma *approx_val_Err*:

$\text{approx_val } G \text{ hp } x \text{ Err}$

lemma *approx_val_Null*:

$\text{approx_val } G \text{ hp } \text{Null } (\text{Ok } (\text{RefT } x))$

lemma *approx_val_imp_approx_val_assConvertible*:

$\llbracket \text{wf_prog } \text{wt } G; \text{approx_val } G \text{ hp } v \text{ (Ok } T); G \vdash T \preceq T' \rrbracket$
 $\implies \text{approx_val } G \text{ hp } v \text{ (Ok } T')$

lemma *approx_val_imp_approx_val_sup_heap*:

$\llbracket \text{approx_val } G \text{ hp } v \text{ at}; \text{hp} \leq_l \text{hp}' \rrbracket \implies \text{approx_val } G \text{ hp}' v \text{ at}$

lemma *approx_val_imp_approx_val_heap_update*:

$\llbracket \text{hp } a = \text{Some } \text{obj}'; G, \text{hp} \vdash v :: \preceq T; \text{obj_ty } \text{obj} = \text{obj_ty } \text{obj}' \rrbracket$
 $\implies G, \text{hp}(a \mapsto \text{obj}) \vdash v :: \preceq T$

lemma *approx_val_imp_approx_val_sup*:

$\llbracket \text{wf_prog } \text{wt } G; \text{approx_val } G \text{ h } v \text{ us}; G \vdash \text{us} \leq_o \text{us}' \rrbracket \implies \text{approx_val } G \text{ h } v \text{ us}'$

```

lemma approx_loc_imp_approx_val_sup:
  [[wf_prog wt G; approx_loc G hp loc LT; idx < length LT; v = loc ! idx;
    G ⊢ LT ! idx ≤ at]]
  ⇒ approx_val G hp v at

lemma approx_loc_Cons:
  approx_loc G hp (s # xs) (l # ls) =
  (approx_val G hp s l ∧ approx_loc G hp xs ls)

lemma assConv_approx_stk_imp_approx_loc:
  [[wf_prog wt G; ∧ tt'. tt' ∈ set (zip tys_n ts) ⇒ tt' ∈ widen G;
    length tys_n = length ts; approx_stk G hp s tys_n]]
  ⇒ approx_loc G hp s (map Ok ts)

lemma approx_loc_imp_approx_loc_sup_heap:
  [[approx_loc G hp lvars lt; hp ≤ hp']] ⇒ approx_loc G hp' lvars lt

lemma approx_loc_imp_approx_loc_sup:
  [[wf_prog wt G; approx_loc G hp lvars lt; G ⊢ lt ≤ lt']]
  ⇒ approx_loc G hp lvars lt'

lemma approx_loc_imp_approx_loc_subst:
  [[approx_loc G hp loc LT; approx_val G hp x X]]
  ⇒ approx_loc G hp (loc[idx := x]) (LT[idx := X])

lemma approx_loc_append:
  length l1 = length l2 ⇒
  approx_loc G hp (l1 @ l2) (l1 @ l2) =
  (approx_loc G hp l1 l1 ∧ approx_loc G hp l2 l2)

lemma approx_stk_rev_lem:
  approx_stk G hp (rev s) (rev t) = approx_stk G hp s t

lemma approx_stk_rev:
  approx_stk G hp (rev s) t = approx_stk G hp s (rev t)

lemma approx_stk_imp_approx_stk_sup_heap:
  [[approx_stk G hp lvars lt; hp ≤ hp']] ⇒ approx_stk G hp' lvars lt

lemma approx_stk_imp_approx_stk_sup:
  [[wf_prog wt G; approx_stk G hp lvars st; G ⊢ map Ok st ≤ map Ok st']]
  ⇒ approx_stk G hp lvars st'

lemma approx_stk_Nil:
  approx_stk G hp [] []

lemma approx_stk_Cons:
  approx_stk G hp (x # stk) (S # ST) =
  (approx_val G hp x (Ok S) ∧ approx_stk G hp stk ST)

lemma approx_stk_Cons_lemma:
  approx_stk G hp stk (S # ST') =
  (∃ s stk'.
    stk = s # stk' ∧ approx_val G hp s (Ok S) ∧ approx_stk G hp stk' ST')

lemma approx_stk_append_lemma:
  approx_stk G hp stk (S @ ST') ⇒
  ∃ s stk'.

```

```

stk = s @ stk' ∧
length s = length S ∧
length stk' = length ST' ∧ approx_stk G hp s S ∧ approx_stk G hp stk' ST'

lemma correct_init_obj:
[[is_class G C; wf_prog wt G]]
⇒ G, h ⊢ (C, map_of (map (λ(f, fT). (f, default_val fT)) (fields (G, C)))) √

lemma oconf_imp_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)) √

lemma oconf_imp_oconf_heap_newref:
[[hp x = None; G, hp ⊢ obj √; G, hp ⊢ obj' √]] ⇒ G, hp(newref hp↦obj') ⊢ obj √

lemma oconf_imp_oconf_heap_update:
[[hp a = Some obj'; obj_ty obj' = obj_ty obj''; G, hp ⊢ obj √]]
⇒ G, hp(a↦obj'') ⊢ obj √

lemma hconf_imp_hconf_newref:
[[hp x = None; G ⊢h hp √; G, hp ⊢ obj √]] ⇒ G ⊢h hp(newref hp↦obj) √

lemma hconf_imp_hconf_field_update:
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))) √

lemma correct_frames_imp_correct_frames_field_update:
[[correct_frames G hp phi rT sig 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)))) phi rT sig frs

lemma correct_frames_imp_correct_frames_newref:
[[hp x = None; correct_frames G hp phi rT sig frs ∧ G, hp ⊢ obj √]]
⇒ correct_frames G (hp(newref hp↦obj)) phi rT sig frs

```

Theory Decl

no theorems

Theory Digest

no theorems

Theory Eval

theorem NewCI:

```

[[new_Addr (fst s) = (a, x);
  s' = c_hupd (fst s(a↦(C, init_vars (fields (G, C)))) (x, s)]]
⇒ G ⊢ Norm s -NewC C>-Addr a-> s'

```

theorem eval_evals_exec_no_xcpt:

```

(G ⊢ (x, s) -e>v-> (x', s') → x' = None → x = None) ∧
(G ⊢ (x, s) -es[>]vs-> (x', s') → x' = None → x = None) ∧
(((x, s), c, x', s') ∈ Eval.exec G → x' = None → x = None)

```

Theory Example**theorem** *not_Object_subcls*: $(Object, C) \in (subcls1\ tprg)^+ \implies R$ **theorem** *subcls_ObjectD*: $tprg \vdash Object \preceq_C C \implies C = Object$ **theorem** *not_Base_subcls_Ext*: $(Base, Ext) \in (subcls1\ tprg)^+ \implies R$ **theorem** *class_tprgD*: $class\ tprg\ C = Some\ z \implies C = Object \vee C = Base \vee C = Ext$ **theorem** *not_class_subcls_class*: $(C, C) \in (subcls1\ tprg)^+ \implies R$ **theorem** *unique_classes*: $unique\ tprg$ **theorem** *subcls_direct*: $class\ G\ C = Some\ (Some\ D, rest) \implies G \vdash C \preceq_C D$ **theorem** *Ext_subcls_Base*: $tprg \vdash Ext \preceq_C Base$ **theorem** *Ext_widen_Base*: $tprg \vdash Class\ Ext \preceq Class\ Base$ **theorem** *acyclic_subcls1_*: $acyclic\ (subcls1\ tprg)$ **theorem** *fields_Object*: $fields\ (tprg, Object) = []$ **theorem** *fields_Base*: $fields\ (tprg, Base) = [((vee, Base), PrimT\ Boolean)]$ **theorem** *fields_Ext*: $fields\ (tprg, Ext) = [((vee, Ext), PrimT\ Integer)] @ fields\ (tprg, Base)$ **theorem** *method_Object*: $method\ (tprg, Object) = map_of\ []$ **theorem** *method_Base*: $method\ (tprg, Base) =$
 $map_of\ [((foo, [Class\ Base]), Base, Class\ Base, foo_Base)]$ **theorem** *method_Ext*: $method\ (tprg, Ext) =$
 $method\ (tprg, Base) ++$
 $map_of\ [((foo, [Class\ Base]), Ext, Class\ Ext, foo_Ext)]$ **theorem** *wf_foo_Base*: $wf_mdecl\ wf_java_mdecl\ tprg\ Base\ ((foo, [Class\ Base]), Class\ Base, foo_Base)$ **theorem** *wf_foo_Ext*: $wf_mdecl\ wf_java_mdecl\ tprg\ Ext\ ((foo, [Class\ Base]), Class\ Ext, foo_Ext)$

```

theorem wf_ObjectC:
wf_cdecl wf_java_mdecl tprg ObjectC

theorem wf_BaseC:
wf_cdecl wf_java_mdecl tprg BaseC

theorem wf_ExtC:
wf_cdecl wf_java_mdecl tprg ExtC

theorem wf_tprg:
wf_prog wf_java_mdecl tprg

theorem appl_methds_foo_Base:
appl_methds tprg Base (foo, [NT]) =
{((Class Base, Class Base), [Class Base])}

theorem max_spec_foo_Base:
max_spec tprg Base (foo, [NT]) = {((Class Base, Class Base), [Class Base])}

theorem wt_test:
(tprg, empty(e ↦
  Class
    Base)) ⊢ Expr
  (e ::= NewC Ext);; Expr (LAcc e..foo({[Class Base]}[Lit Null])) √

theorem exec_test:
new_Addr (fst (snd s0)) = (a, None) ⇒ (s0, test, s3) ∈ Eval.exec tprg

```

Theory JBasis

```

theorem image_rev:
x ∈ f ‘‘ A ⇒ ∃y. y ∈ A ∧ x = f y

theorem some_subset_the:
{y. x = Some y} ⊆ {the x}

theorem fst_in_set_lemma:
(x, y) ∈ set l ⇒ x ∈ fst ‘‘ set l

theorem unique_Nil:
unique []

theorem unique_Cons:
unique ((x, y) # l) = (unique l ∧ (∀y. (x, y) ∉ set l))

theorem unique_append:
[[unique l'; unique l; ∀(x, y) ∈ set l. ∀(x', y') ∈ set l'. x' ≠ x]]
⇒ unique (l @ l')

theorem unique_map_inj:
[[unique l; inj f]] ⇒ unique (map (λ(k, x). (f k, g k x)) l)

theorem unique_map_Pair:
unique l ⇒ unique (map (split (λk. Pair (k, C))) l)

theorem image_cong:
[[M = N; ∧ x. x ∈ N ⇒ f x = g x]] ⇒ f ‘‘ M = g ‘‘ N

```

theorem unique_map_of_Some_conv:

unique xys $\implies$ (map_of xys x = Some y) = ((x, y) $\in$ set xys)

theorem Ball_set_table:

Ball (set l) (split P) $\implies \forall x y. \text{map_of } l \ x = \text{Some } y \longrightarrow P \ x \ y$

theorem map_of_map:

map_of (map ($\lambda(a, b). (a, f \ b)$) xs) x = option_map f (map_of xs x)

Theory JTypeSafe

theorem NewC_conforms:

$\llbracket h \ a = \text{None}; (h, l) :: \preceq (G, lT); \text{wf_prog } \text{wf_mb } G; \text{is_class } G \ C \rrbracket$
 $\implies (h(a \mapsto (C, \text{init_vars } (\text{fields } (G, C))))), l) :: \preceq (G, lT)$

theorem Cast_conf:

$\llbracket \text{wf_prog } \text{wf_mb } G; G, h \vdash v :: \preceq \text{Class } C; G \vdash C \preceq? D; \text{cast_ok } G \ D \ h \ v \rrbracket$
 $\implies G, h \vdash v :: \preceq \text{Class } D$

theorem FAcc_type_sound:

$\llbracket \text{wf_prog } \text{wf_mb } G; \text{field } (G, C) \text{ fn} = \text{Some } (fd, ft); (h, l) :: \preceq (G, lT);$
 $x' = \text{None} \longrightarrow G, h \vdash a' :: \preceq \text{Class } C; \text{np } a' \ x' = \text{None} \rrbracket$
 $\implies G, h \vdash \text{the } (\text{snd } (\text{the } (h \ (\text{the_Addr } a')))) \ (fn, fd) :: \preceq ft$

theorem FAss_type_sound:

$\llbracket \text{wf_prog } \text{wf_mb } G; a = \text{the_Addr } a'; (c, fs) = \text{the } (h \ a); (G, lT) \vdash v :: T';$
 $G \vdash T' \preceq ft; (G, lT) \vdash aa :: \text{Class } C; \text{field } (G, C) \text{ fn} = \text{Some } (fd, ft);$
 $h'' \leq | h'; x' = \text{None} \longrightarrow G, h' \vdash a' :: \preceq \text{Class } C; h' \leq | h;$
 $(h, l) :: \preceq (G, lT); G, h \vdash x :: \preceq T'; \text{np } a' \ x' = \text{None} \rrbracket$
 $\implies h'' \leq | h(a \mapsto (c, fs((fn, fd) \mapsto x))) \wedge$
 $(h(a \mapsto (c, fs((fn, fd) \mapsto x))), l) :: \preceq (G, lT) \wedge G, h(a \mapsto$
 $(c, fs((fn, fd) \mapsto x))) \vdash x :: \preceq T'$

theorem Call_lemma2:

$\llbracket \text{wf_prog } \text{wf_mb } G; \text{list_all2 } (\text{conf } G \ h) \text{ pvs } pTs;$
 $\text{list_all2 } (\lambda T \ T'. G \vdash T \preceq T') \text{ pTs } pTs'; \text{wf_mhead } G \ (mn, pTs') \text{ rT};$
 $\text{length } pTs' = \text{length } pns; \text{nodups } pns;$
 $\text{Ball } (\text{set } lvars) (\text{split } (\lambda vn. \text{is_type } G)) \rrbracket$
 $\implies G, h \vdash \text{init_vars } lvars(pns[\mapsto]pvs) [:: \preceq] \text{map_of } lvars(pns[\mapsto]pTs')$

theorem Call_type_sound:

$\llbracket \text{wf_java_prog } G; a' \neq \text{Null}; (h, l) :: \preceq (G, lT);$
 $\text{max_spec } G \ C \ (mn, pTsa) = \{((mda, rTa), pTs')\}; xc \leq | xh; xh \leq | h;$
 $\text{list_all2 } (\text{conf } G \ h) \text{ pvs } pTsa;$
 $(md, rT, pns, lvars, blk, res) =$
 $\text{the } (\text{method } (G, \text{fst } (\text{the } (h \ (\text{the_Addr } a'))))) \ (mn, pTs'));$
 $\forall lT. (h, \text{init_vars } lvars(pns[\mapsto]pvs)(\text{This} \mapsto a')) :: \preceq (G, lT) \longrightarrow$
 $(G, lT) \vdash \text{blk } \checkmark \longrightarrow h \leq | xi \wedge (xi, xl) :: \preceq (G, lT);$
 $\forall lT. (xi, xl) :: \preceq (G, lT) \longrightarrow$
 $(\forall T. (G, lT) \vdash \text{res} :: T \longrightarrow$
 $xi \leq | h' \wedge$
 $(h', xj) :: \preceq (G, lT) \wedge (x' = \text{None} \longrightarrow G, h' \vdash v :: \preceq T));$
 $G, xh \vdash a' :: \preceq \text{Class } C \rrbracket$
 $\implies xc \leq | h' \wedge (h', l) :: \preceq (G, lT) \wedge (x' = \text{None} \longrightarrow G, h' \vdash v :: \preceq rTa)$

theorem eval_evals_exec_type_sound:

$$\begin{aligned}
& \text{wf_java_prog } G \implies \\
& (((x, h, l), e, v, x', h', l') \in \text{eval } G \longrightarrow \\
& \quad (\forall lT. (h, l) :: \preceq (G, lT) \longrightarrow \\
& \quad \quad (\forall T. (G, lT) \vdash e :: T \longrightarrow \\
& \quad \quad \quad h \leq | h' \wedge \\
& \quad \quad \quad (h', l') :: \preceq (G, lT) \wedge (x' = \text{None} \longrightarrow G, h' \vdash v :: \preceq T)))) \wedge \\
& (((x, h, l), es, vs, x', h', l') \in \text{evals } G \longrightarrow \\
& \quad (\forall lT. (h, l) :: \preceq (G, lT) \longrightarrow \\
& \quad \quad (\forall Ts. (G, lT) \vdash es [::] Ts \longrightarrow \\
& \quad \quad \quad h \leq | h' \wedge \\
& \quad \quad \quad (h', l') :: \preceq (G, lT) \wedge \\
& \quad \quad \quad (x' = \text{None} \longrightarrow \text{list_all2 } (\text{conf } G \ h') \text{ vs } Ts)))) \wedge \\
& (((x, h, l), c, x', h', l') \in \text{Eval.exec } G \longrightarrow \\
& \quad (\forall lT. (h, l) :: \preceq (G, lT) \longrightarrow (G, lT) \vdash c \checkmark \longrightarrow h \leq | h' \wedge (h', l') :: \preceq (G, lT)))
\end{aligned}$$

theorem eval_type_sound:

$$\begin{aligned}
& \llbracket G = \text{fst } E; \text{wf_java_prog } G; G \vdash (x, s) \text{-e>v-}> (x', s'); s :: \preceq E; E \vdash e :: T \rrbracket \\
& \implies s' :: \preceq E \wedge (x' = \text{None} \longrightarrow G, \text{fst } s' \vdash v :: \preceq T)
\end{aligned}$$

theorem exec_type_sound:

$$\begin{aligned}
& \llbracket G = \text{fst } E; \text{wf_java_prog } G; ((x, s), s0, x', s') \in \text{Eval.exec } G; s :: \preceq E; \\
& \quad E \vdash s0 \checkmark \rrbracket \\
& \implies s' :: \preceq E
\end{aligned}$$

theorem all_methods_understood:

$$\begin{aligned}
& \llbracket G = \text{fst } E; \text{wf_java_prog } G; G \vdash (x, s) \text{-e>a'-}> \text{Norm } s'; a' \neq \text{Null}; s :: \preceq E; \\
& \quad E \vdash e :: \text{Class } C; \text{method } (G, C) \text{ sig} \neq \text{None} \rrbracket \\
& \implies \text{method } (G, \text{fst } (\text{the } (\text{fst } s' (\text{the_Addr } a')))) \text{ sig} \neq \text{None}
\end{aligned}$$

Theory JVMExec

no theorems

Theory JVMExecInstr

no theorems

Theory JVMInstructions

no theorems

Theory JVMState

no theorems

Theory LBVComplete

lemma make_cert_target:

$$\llbracket pc < \text{length } ins; \text{is_target } ins \ pc \rrbracket \implies \text{make_cert } ins \ \text{phi} \ ! \ pc = \text{phi} \ ! \ pc$$

lemma *make_cert_approx*:

$\llbracket pc < \text{length ins}; \text{make_cert ins } \phi ! pc \neq \phi ! pc \rrbracket$
 $\implies \text{make_cert ins } \phi ! pc = \text{None}$

lemma *make_cert_contains_targets*:

$pc < \text{length ins} \implies \text{contains_targets ins (make_cert ins } \phi) \phi pc$

theorem *fits_make_cert*:

$\text{LBVComplete.fits ins (make_cert ins } \phi) \phi$

lemma *fitsD*:

$\llbracket \text{LBVComplete.fits ins cert } \phi; pc' \in \text{set (succs (ins ! pc) pc)};$
 $pc' \neq \text{Suc pc}; pc < \text{length ins}; pc' < \text{length ins} \rrbracket$
 $\implies \text{cert ! } pc' = \phi ! pc'$

lemma *fitsD2*:

$\llbracket \text{LBVComplete.fits ins cert } \phi; pc < \text{length ins}; \text{cert ! } pc = \text{Some } s \rrbracket$
 $\implies \text{cert ! } pc = \phi ! pc$

lemma *wtl_inst_mono*:

$\llbracket \text{wtl_inst } i \text{ } G \text{ } rT \text{ } s1 \text{ cert mpc } pc = Ok \text{ } s1'; \text{LBVComplete.fits ins cert } \phi;$
 $pc < \text{length ins}; G \vdash s2 \leq' s1; i = \text{ins ! } pc \rrbracket$
 $\implies \exists s2'. \text{wtl_inst (ins ! pc) } G \text{ } rT \text{ } s2 \text{ cert mpc } pc = Ok \text{ } s2' \wedge G \vdash s2' \leq' s1'$

lemma *wtl_cert_mono*:

$\llbracket \text{wtl_cert } i \text{ } G \text{ } rT \text{ } s1 \text{ cert mpc } pc = Ok \text{ } s1'; \text{LBVComplete.fits ins cert } \phi;$
 $pc < \text{length ins}; G \vdash s2 \leq' s1; i = \text{ins ! } pc \rrbracket$
 $\implies \exists s2'. \text{wtl_cert (ins ! pc) } G \text{ } rT \text{ } s2 \text{ cert mpc } pc = Ok \text{ } s2' \wedge G \vdash s2' \leq' s1'$

lemma *wt_instr_imp_wtl_inst*:

$\llbracket \text{wt_instr (ins ! pc) } G \text{ } rT \text{ } \phi \text{ max_pc } pc; \text{LBVComplete.fits ins cert } \phi;$
 $pc < \text{length ins}; \text{length ins} = \text{max_pc} \rrbracket$
 $\implies \text{wtl_inst (ins ! pc) } G \text{ } rT \text{ } (\phi ! pc) \text{ cert max_pc } pc \neq \text{Err}$

lemma *wt_less_wtl*:

$\llbracket \text{wt_instr (ins ! pc) } G \text{ } rT \text{ } \phi \text{ max_pc } pc;$
 $\text{wtl_inst (ins ! pc) } G \text{ } rT \text{ } (\phi ! pc) \text{ cert max_pc } pc = Ok \text{ } s;$
 $\text{LBVComplete.fits ins cert } \phi; \text{Suc pc} < \text{length ins}; \text{length ins} = \text{max_pc} \rrbracket$
 $\implies G \vdash s \leq' \phi ! \text{Suc pc}$

lemma *wt_instr_imp_wtl_cert*:

$\llbracket \text{wt_instr (ins ! pc) } G \text{ } rT \text{ } \phi \text{ max_pc } pc; \text{LBVComplete.fits ins cert } \phi;$
 $pc < \text{length ins}; \text{length ins} = \text{max_pc} \rrbracket$
 $\implies \text{wtl_cert (ins ! pc) } G \text{ } rT \text{ } (\phi ! pc) \text{ cert max_pc } pc \neq \text{Err}$

lemma *wt_less_wtl_cert*:

$\llbracket \text{wt_instr (ins ! pc) } G \text{ } rT \text{ } \phi \text{ max_pc } pc;$
 $\text{wtl_cert (ins ! pc) } G \text{ } rT \text{ } (\phi ! pc) \text{ cert max_pc } pc = Ok \text{ } s;$
 $\text{LBVComplete.fits ins cert } \phi; \text{Suc pc} < \text{length ins}; \text{length ins} = \text{max_pc} \rrbracket$
 $\implies G \vdash s \leq' \phi ! \text{Suc pc}$

theorem *wt_imp_wtl_inst_list*:

$\forall pc. (\forall pc'. pc' < \text{length all_ins} \longrightarrow$
 $\quad \text{wt_instr (all_ins ! pc') } G \text{ } rT \text{ } \phi \text{ (length all_ins) } pc') \longrightarrow$
 $\quad \text{LBVComplete.fits all_ins cert } \phi \longrightarrow$
 $\quad (\exists l. pc = \text{length } l \wedge \text{all_ins} = l @ \text{ins}) \longrightarrow$
 $\quad pc < \text{length all_ins} \longrightarrow$

$$(\forall s. G \vdash s \leq' \text{phi} \text{ ! } pc \longrightarrow \\ \text{wtl_inst_list ins } G \text{ rT cert (length all_ins) } pc \text{ s} \neq \text{Err})$$

lemma fits_imp_wtl_method_complete:

$$\llbracket \text{wt_method } G \text{ C pTs rT mxl ins phi; LBVComplete.fits ins cert phi} \rrbracket \\ \implies \text{wtl_method } G \text{ C pTs rT mxl ins cert}$$

lemma wtl_method_complete:

$$\text{wt_method } G \text{ C pTs rT mxl ins phi} \implies \\ \text{wtl_method } G \text{ C pTs rT mxl ins (make_cert ins phi)}$$

lemma unique_set:

$$(a, b, c, d) \in \text{set } l \longrightarrow \\ \text{unique } l \longrightarrow (a', b', c', d') \in \text{set } l \longrightarrow a = a' \longrightarrow b = b' \wedge c = c' \wedge d = d'$$

lemma unique_epsilon:

$$(a, b, c, d) \in \text{set } l \longrightarrow \\ \text{unique } l \longrightarrow \\ (\text{SOME } (a', b', c', d')). (a', b', c', d') \in \text{set } l \wedge a' = a = (a, b, c, d)$$

theorem wtl_complete:

$$\text{wt_jvm_prog } G \text{ Phi} \implies \text{wtl_jvm_prog } G \text{ (make_Cert } G \text{ Phi)}$$

Theory LBVCorrect

lemma fitsD_None:

$$\llbracket \text{LBVCorrect.fits phi is } G \text{ rT s0 cert; } pc < \text{length is;} \\ \text{wtl_inst_list (take } pc \text{ is) } G \text{ rT cert (length is) } 0 \text{ s0} = 0k \text{ s1;} \\ \text{cert ! } pc = \text{None} \rrbracket \\ \implies \text{phi ! } pc = s1$$

lemma fitsD_Some:

$$\llbracket \text{LBVCorrect.fits phi is } G \text{ rT s0 cert; } pc < \text{length is;} \\ \text{wtl_inst_list (take } pc \text{ is) } G \text{ rT cert (length is) } 0 \text{ s0} = 0k \text{ s1;} \\ \text{cert ! } pc = \text{Some } t \rrbracket \\ \implies \text{phi ! } pc = \text{Some } t$$

lemma make_phi_Some:

$$\llbracket pc < \text{length is; cert ! } pc = \text{Some } t \rrbracket \\ \implies \text{make_phi is } G \text{ rT s0 cert ! } pc = \text{Some } t$$

lemma make_phi_None:

$$\llbracket pc < \text{length is; cert ! } pc = \text{None} \rrbracket \\ \implies \text{make_phi is } G \text{ rT s0 cert ! } pc = \\ \text{val (wtl_inst_list (take } pc \text{ is) } G \text{ rT cert (length is) } 0 \text{ s0)}$$

lemma exists_phi:

$$\exists \text{phi. LBVCorrect.fits phi is } G \text{ rT s0 cert}$$

lemma fits_lemma1:

$$\llbracket \text{wtl_inst_list is } G \text{ rT cert (length is) } 0 \text{ s} = 0k \text{ s'}; \\ \text{LBVCorrect.fits phi is } G \text{ rT s cert} \rrbracket \\ \implies \forall pc \text{ t. } pc < \text{length is} \longrightarrow \text{cert ! } pc = \text{Some } t \longrightarrow \text{phi ! } pc = \text{Some } t$$

lemma wtl_suc_pc:

$$\llbracket \text{wtl_inst_list is } G \text{ rT cert (length is) } 0 \text{ s} \neq \text{Err;} \\$$

```

    wtl_inst_list (take pc is) G rT cert (length is) 0 s = Ok s';
    wtl_cert (is ! pc) G rT s' cert (length is) pc = Ok s'';
    LBVCorrect.fits phi is G rT s cert; Suc pc < length is
  ==> G ⊢ s'' <= phi ! Suc pc

```

lemma wtl_fits_wt:

```

  [[wtl_inst_list is G rT cert (length is) 0 s ≠ Err;
    LBVCorrect.fits phi is G rT s cert; pc < length is]]
  ==> wt_instr (is ! pc) G rT phi (length is) pc

```

lemma fits_first:

```

  [[0 < length is; wtl_inst_list is G rT cert (length is) 0 s ≠ Err;
    LBVCorrect.fits phi is G rT s cert]]
  ==> G ⊢ s <= phi ! 0

```

lemma wtl_method_correct:

```

  wtl_method G C pTs rT mxl ins cert ==> ∃phi. wt_method G C pTs rT mxl ins phi

```

lemma unique_set:

```

  (a, b, c, d) ∈ set l →
  unique l → (a', b', c', d') ∈ set l → a = a' → b = b' ∧ c = c' ∧ d = d'

```

lemma unique_epsilon:

```

  (a, b, c, d) ∈ set l →
  unique l →
  (SOME (a', b', c', d')). (a', b', c', d') ∈ set l ∧ a' = a) = (a, b, c, d)

```

theorem wtl_correct:

```

  wtl_jvm_prog G cert ==> ∃Phi. wt_jvm_prog G Phi

```

Theory LBVSpec

lemma wtl_inst_0k:

```

  (wtl_inst i G rT s cert max_pc pc = Ok s') =
  (app i G rT s ∧
   (∀pc'∈set (succs i pc).
    pc' < max_pc ∧ (pc' ≠ pc + 1 → G ⊢ step i G s <= cert ! pc')) ∧
   (if pc + 1 ∈ set (succs i pc) then s' = step i G s
    else s' = cert ! (pc + 1)))

```

lemma strict_Some:

```

  (strict f x = Ok y) = (∃z. x = Ok z ∧ f z = Ok y)

```

lemma wtl_Cons:

```

  (wtl_inst_list (i # is) G rT cert max_pc pc s ≠ Err) =
  (∃s'. wtl_cert i G rT s cert max_pc pc = Ok s' ∧
   wtl_inst_list is G rT cert max_pc (pc + 1) s' ≠ Err)

```

lemma wtl_append:

```

  ∀s pc.
  (wtl_inst_list (a @ b) G rT cert mpc pc s = Ok s') =
  (∃s''. wtl_inst_list a G rT cert mpc pc s = Ok s'' ∧
   wtl_inst_list b G rT cert mpc (pc + length a) s'' = Ok s')

```

lemma wtl_take:

wtl_inst_list is G rT cert mpc pc s = Ok s'' $\implies$
 $\exists s'. \text{ wtl_inst_list } (\text{take } pc' \text{ is}) \text{ G rT cert mpc pc s} = \text{Ok } s'$

lemma take_Suc:

$\forall n. n < \text{length } l \implies \text{take } (\text{Suc } n) \text{ l} = \text{take } n \text{ l} @ [l ! n]$

lemma wtl_Suc:

$\llbracket \text{wtl_inst_list } (\text{take } pc \text{ is}) \text{ G rT cert } (\text{length is}) \text{ 0 s} = \text{Ok } s';$
 $\text{ wtl_cert } (is ! pc) \text{ G rT s' cert } (\text{length is}) \text{ pc} = \text{Ok } s'';$
 $\text{ Suc } pc < \text{length is} \rrbracket$
 $\implies \text{ wtl_inst_list } (\text{take } (\text{Suc } pc) \text{ is}) \text{ G rT cert } (\text{length is}) \text{ 0 s} = \text{Ok } s''$

lemma wtl_all:

$\llbracket \text{wtl_inst_list is G rT cert } (\text{length is}) \text{ 0 s} \neq \text{Err}; pc < \text{length is} \rrbracket$
 $\implies \exists s' s''.$
 $\text{ wtl_inst_list } (\text{take } pc \text{ is}) \text{ G rT cert } (\text{length is}) \text{ 0 s} = \text{Ok } s' \wedge$
 $\text{ wtl_cert } (is ! pc) \text{ G rT s' cert } (\text{length is}) \text{ pc} = \text{Ok } s''$

Theory State

theorem np_raise_if:

np Null (raise_if c xc None) = Some (if c then xc else NP)

Theory Step

lemma 1:

$2 < \text{length } a \implies \exists l \text{ l' l'' ls. } a = l \# l' \# l'' \# ls$

lemma 2:

$\neg 2 < \text{length } a \implies a = [] \vee (\exists l. a = [l]) \vee (\exists l \text{ l'}. a = [l, l'])$

lemma appNone:

app i G rT None = True

lemma appLoad:

app (Load idx) G rT (Some s) = (idx < length (snd s) $\wedge$ snd s ! idx $\neq$ Err)

lemma appStore:

app (Store idx) G rT (Some s) =
 $(\exists ts \text{ ST LT. } s = (ts \# \text{ST}, \text{LT}) \wedge \text{idx} < \text{length LT})$

lemma appBipush:

app (Bipush i) G rT (Some s) = True

lemma appAconst:

app Aconst_null G rT (Some s) = True

lemma appGetField:

app (Getfield F C) G rT (Some s) =
 $(\exists oT \text{ vT ST LT.}$
 $\text{ s} = (oT \# \text{ST}, \text{LT}) \wedge$
 $\text{is_class } G \text{ C} \wedge \text{field } (G, C) \text{ F} = \text{Some } (C, \text{vT}) \wedge G \vdash oT \preceq \text{Class } C)$

lemma appPutField:

```
app (Putfield F C) G rT (Some s) =
  (∃ vT vT' oT ST LT.
    s = (vT # oT # ST, LT) ∧
    is_class G C ∧
    field (G, C) F = Some (C, vT')) ∧ G ⊢ oT ≤ Class C ∧ G ⊢ vT ≤ vT')
```

lemma appNew:

```
app (New C) G rT (Some s) = is_class G C
```

lemma appCheckcast:

```
app (Checkcast C) G rT (Some s) =
  (∃ rT ST LT. s = (RefT rT # ST, LT) ∧ is_class G C)
```

lemma appPop:

```
app Pop G rT (Some s) = (∃ ts ST LT. s = (ts # ST, LT))
```

lemma appDup:

```
app Dup G rT (Some s) = (∃ ts ST LT. s = (ts # ST, LT))
```

lemma appDup_x1:

```
app Dup_x1 G rT (Some s) = (∃ ts1 ts2 ST LT. s = (ts1 # ts2 # ST, LT))
```

lemma appDup_x2:

```
app Dup_x2 G rT (Some s) =
  (∃ ts1 ts2 ts3 ST LT. s = (ts1 # ts2 # ts3 # ST, LT))
```

lemma appSwap:

```
app Swap G rT (Some s) = (∃ ts1 ts2 ST LT. s = (ts1 # ts2 # ST, LT))
```

lemma appIAdd:

```
app IAdd G rT (Some s) =
  (∃ ST LT. s = (PrimT Integer # PrimT Integer # ST, LT))
```

lemma appIfcmpeq:

```
app (Ifcmpeq b) G rT (Some s) =
  (∃ ts1 ts2 ST LT.
    s = (ts1 # ts2 # ST, LT) ∧
    ((∃ p. ts1 = PrimT p ∧ ts2 = PrimT p) ∨
     (∃ r r'. ts1 = RefT r ∧ ts2 = RefT r'))))
```

lemma appReturn:

```
app Return G rT (Some s) = (∃ T ST LT. s = (T # ST, LT) ∧ G ⊢ T ≤ rT)
```

lemma appGoto:

```
app (Goto branch) G rT (Some s) = True
```

lemma appInvoke:

```
app (Invoke C mn fpTs) G rT (Some s) =
  (∃ apTs X ST LT mD' rT' b'.
    s = (rev apTs @ X # ST, LT) ∧
    length apTs = length fpTs ∧
    G ⊢ X ≤ Class C ∧
    (∀ (aT, fT) ∈ set (zip apTs fpTs). G ⊢ aT ≤ fT) ∧
    method (G, C) (mn, fpTs) = Some (mD', rT', b'))
```

lemma step_Some:

```
step i G (Some s) ≠ None
```

lemma *step_None*:
 $\text{step } i \ G \ \text{None} = \text{None}$

Theory StepMono

lemma *PrimT_PrimT*:
 $G \vdash xb \preceq \text{PrimT } p = (xb = \text{PrimT } p)$

lemma *sup_loc_some*:
 $\llbracket G \vdash b \leq_l y; n < \text{length } y; y ! n = \text{Ok } t \rrbracket$
 $\implies \exists t. b ! n = \text{Ok } t \wedge G \vdash b ! n \leq_o y ! n$

lemma *all_widen_is_sup_loc*:
 $\forall b. \text{length } a = \text{length } b \longrightarrow$
 $(\forall x \in \text{set } (\text{zip } a \ b). x \in \text{widen } G) = G \vdash \text{map } \text{Ok } a \leq_l \text{map } \text{Ok } b$

lemma *append_length_n*:
 $n \leq \text{length } x \implies \exists a \ b. x = a @ b \wedge \text{length } a = n$

lemma *rev_append_cons*:
 $n < \text{length } x \implies \exists a \ b \ c. x = \text{rev } a @ b \# c \wedge \text{length } a = n$

lemma *app_mono*:
 $\llbracket G \vdash s \leq' s'; \text{app } i \ G \ rT \ s' \rrbracket \implies \text{app } i \ G \ rT \ s$

lemma *step_mono_Some*:
 $\llbracket \text{succs } i \ pc \neq []; \text{app } i \ G \ rT \ (\text{Some } s2); G \vdash s1 \leq_s s2 \rrbracket$
 $\implies G \vdash \text{the } (\text{step } i \ G \ (\text{Some } s1)) \leq_s \text{the } (\text{step } i \ G \ (\text{Some } s2))$

lemma *step_mono*:
 $\llbracket \text{succs } i \ pc \neq []; \text{app } i \ G \ rT \ s2; G \vdash s1 \leq' s2 \rrbracket$
 $\implies G \vdash \text{step } i \ G \ s1 \leq' \text{step } i \ G \ s2$

Theory Store

theorem *newref_None*:
 $hp \ x = \text{None} \implies hp \ (\text{newref } hp) = \text{None}$

Theory Term

no theorems

Theory Type

no theorems

Theory TypeRel

theorem *finite_subcls1*:

finite (*subcls1* *G*)

theorem *subcls_is_class*:

$(C, D) \in (\text{subcls1 } G)^+ \implies \text{is_class } G \ C$

theorem *wf_rel_lemma*:

$\text{wf } \{(A, x), (B, y). A = B \wedge \text{wf } (R \ A) \wedge (x, y) \in R \ A\}$

theorem *wf_subcls1_rel*:

wf *subcls1_rel*

theorem *widen_PrimT_RefT*:

$G \vdash \text{PrimT } pT \preceq \text{RefT } rT = \text{False}$

theorem *widen_RefT*:

$G \vdash \text{RefT } R \preceq T \implies \exists t. T = \text{RefT } t$

theorem *widen_RefT2*:

$G \vdash S \preceq \text{RefT } R \implies \exists t. S = \text{RefT } t$

theorem *widen_Class*:

$G \vdash \text{Class } C \preceq T \implies \exists D. T = \text{Class } D$

theorem *widen_Class_NullT*:

$G \vdash \text{Class } C \preceq \text{NT} = \text{False}$

theorem *widen_Class_Class*:

$G \vdash \text{Class } C \preceq \text{Class } D = G \vdash C \preceq C \ D$

theorem *widen_trans*:

$\llbracket G \vdash S \preceq U; G \vdash U \preceq T \rrbracket \implies G \vdash S \preceq T$

Theory Value

no theorems

Theory WellForm

theorem *subcls1_wfD*:

$\llbracket G \vdash C \prec_{C1} D; \text{wf_prog } \text{wf_mb } G \rrbracket \implies D \neq C \wedge (D, C) \notin (\text{subcls1 } G)^+$

theorem *subcls_asym*:

$\llbracket \text{wf_prog } \text{wf_mb } G; (C, D) \in (\text{subcls1 } G)^+ \rrbracket \implies (D, C) \notin (\text{subcls1 } G)^+$

theorem *subcls_induct*:

$\llbracket \text{wf_prog } \text{wf_mb } G; \bigwedge C. \forall D. (C, D) \in (\text{subcls1 } G)^+ \longrightarrow P \ D \implies P \ C \rrbracket \implies P \ C$

theorem *subcls1_induct*:

$\llbracket \text{is_class } G \ C; \text{wf_prog } \text{wf_mb } G; P \ \text{Object};$
 $\bigwedge C \ D \ fs \ ms.$
 $\llbracket C \neq \text{Object}; \text{is_class } G \ C;$
 $\text{class } G \ C = \text{Some } (\text{Some } D, fs, ms) \wedge$
 $\text{wf_cdecl } \text{wf_mb } G \ (C, \text{Some } D, fs, ms) \wedge$

$$\begin{aligned}
& G \vdash C \prec_{C1} D \wedge \text{is_class } G D \wedge P D \\
& \implies P C \\
& \implies P C
\end{aligned}$$

theorem method_rec_lemma:

$$\begin{aligned}
& \llbracket \text{wf } ((\text{subcls1 } G)^{-1}); \\
& \quad \forall D \text{ fs ms. class } G C = \text{Some } (\text{Some } D, \text{fs}, \text{ms}) \longrightarrow \text{is_class } G D \rrbracket \\
& \implies \text{method } (G, C) = \\
& \quad (\text{case class } G C \text{ of None } \Rightarrow \text{empty} \\
& \quad \mid \text{Some } (sc, \text{fs}, \text{ms}) \Rightarrow \\
& \quad \quad (\text{case } sc \text{ of None } \Rightarrow \text{empty} \mid \text{Some } D \Rightarrow \text{method } (G, D)) ++ \\
& \quad \quad \text{map_of } (\text{map } (\lambda(s, m). (s, C, m)) \text{ ms}))
\end{aligned}$$

theorem method_rec:

$$\begin{aligned}
& \text{wf_prog wf_mb } G \implies \\
& \text{method } (G, C) = \\
& (\text{case class } G C \text{ of None } \Rightarrow \text{empty} \\
& \mid \text{Some } (sc, \text{fs}, \text{ms}) \Rightarrow \\
& \quad (\text{case } sc \text{ of None } \Rightarrow \text{empty} \mid \text{Some } D \Rightarrow \text{method } (G, D)) ++ \\
& \quad \text{map_of } (\text{map } (\lambda(s, m). (s, C, m)) \text{ ms}))
\end{aligned}$$

theorem fields_rec_lemma:

$$\begin{aligned}
& \llbracket \text{wf } ((\text{subcls1 } G)^{-1}); \text{class } G C = \text{Some } (sc, \text{fs}, \text{ms}); \\
& \quad \forall C. sc = \text{Some } C \longrightarrow \text{is_class } G C \rrbracket \\
& \implies \text{fields } (G, C) = \\
& \quad \text{map } (\text{split } (\lambda \text{fn. Pair } (\text{fn}, C))) \text{ fs } @ \\
& \quad (\text{case } sc \text{ of None } \Rightarrow [] \mid \text{Some } D \Rightarrow \text{fields } (G, D))
\end{aligned}$$

theorem fields_rec:

$$\begin{aligned}
& \llbracket \text{class } G C = \text{Some } (sc, \text{fs}, \text{ms}); \text{wf_prog wf_mb } G \rrbracket \\
& \implies \text{fields } (G, C) = \\
& \quad \text{map } (\text{split } (\lambda \text{fn. Pair } (\text{fn}, C))) \text{ fs } @ \\
& \quad (\text{case } sc \text{ of None } \Rightarrow [] \mid \text{Some } D \Rightarrow \text{fields } (G, D))
\end{aligned}$$

theorem subcls_C_Object:

$$\llbracket \text{is_class } G C; \text{wf_prog wf_mb } G \rrbracket \implies G \vdash C \preceq_C \text{Object}$$

theorem fields_mono:

$$\begin{aligned}
& \llbracket (C', C) \in (\text{subcls1 } G)^{+}; \text{wf_prog wf_mb } G; x \in \text{set } (\text{fields } (G, C)) \rrbracket \\
& \implies x \in \text{set } (\text{fields } (G, C'))
\end{aligned}$$

theorem widen_fields_defpl':

$$\begin{aligned}
& \llbracket \text{is_class } G C; \text{wf_prog wf_mb } G \rrbracket \\
& \implies \forall ((\text{fn}, \text{fd}), \text{fT}) \in \text{set } (\text{fields } (G, C)). G \vdash C \preceq_C \text{fd}
\end{aligned}$$

theorem widen_fields_defpl:

$$\begin{aligned}
& \llbracket \text{is_class } G C; \text{wf_prog wf_mb } G; ((\text{fn}, \text{fd}), \text{fT}) \in \text{set } (\text{fields } (G, C)) \rrbracket \\
& \implies G \vdash C \preceq_C \text{fd}
\end{aligned}$$

theorem unique_fields:

$$\llbracket \text{is_class } G C; \text{wf_prog wf_mb } G \rrbracket \implies \text{unique } (\text{fields } (G, C))$$

theorem widen_fields_mono:

$$\begin{aligned}
& \llbracket \text{wf_prog wf_mb } G; G \vdash C' \preceq_C C; \text{map_of } (\text{fields } (G, C)) f = \text{Some } \text{ft} \rrbracket \\
& \implies \text{map_of } (\text{fields } (G, C')) f = \text{Some } \text{ft}
\end{aligned}$$

theorem widen_cfs_fields:

$\llbracket \text{field } (G, C) \text{ fn} = \text{Some } (fd, fT); G \vdash C' \preceq_C C; \text{wf_prog wf_mb } G \rrbracket$
 $\implies \text{map_of } (\text{fields } (G, C')) (fn, fd) = \text{Some } fT$

theorem method_wf_mdecl:

$\llbracket \text{wf_prog wf_mb } G; \text{method } (G, C) \text{ sig} = \text{Some } (md, mh, m) \rrbracket$
 $\implies G \vdash C \preceq_C md \wedge \text{wf_mdecl wf_mb } G \text{ md } (sig, mh, m)$

theorem subcls_widen_methd:

$\llbracket G \vdash T \preceq_C T'; \text{wf_prog wf_mb } G; \text{method } (G, T') \text{ sig} = \text{Some } (D, rT, b) \rrbracket$
 $\implies \exists D' \ rT' \ b'. \text{method } (G, T) \text{ sig} = \text{Some } (D', rT', b') \wedge G \vdash rT' \preceq rT$

theorem subtype_widen_methd:

$\llbracket G \vdash C \preceq_C D; \text{wf_prog wf_mb } G; \text{method } (G, D) \text{ sig} = \text{Some } (md, rT, b) \rrbracket$
 $\implies \exists md' \ rT' \ b'. \text{method } (G, C) \text{ sig} = \text{Some } (md', rT', b') \wedge G \vdash rT' \preceq rT$

theorem method_in_md:

$\llbracket \text{wf_prog wf_mb } G; \text{method } (G, C) \text{ sig} = \text{Some } (D, mh, code) \rrbracket$
 $\implies \text{is_class } G \ D \wedge \text{method } (G, D) \text{ sig} = \text{Some } (D, mh, code)$

theorem is_type_fields:

$\llbracket \text{is_class } G \ C; \text{wf_prog wf_mb } G; x \in \text{set } (\text{fields } (G, C)) \rrbracket \implies \text{is_type } G \ (\text{snd } x)$

Theory WellType

theorem widen_methd:

$\llbracket \text{method } (G, C) \text{ sig} = \text{Some } (md, rT, b); \text{wf_prog wf_mb } G; G \vdash T'' \preceq_C C \rrbracket$
 $\implies \exists md' \ rT' \ b'. \text{method } (G, T'') \text{ sig} = \text{Some } (md', rT', b') \wedge G \vdash rT' \preceq rT$

theorem Call_lemma:

$\llbracket \text{method } (G, C) \text{ sig} = \text{Some } (md, rT, b); G \vdash T'' \preceq_C C; \text{wf_prog wf_mb } G \rrbracket$
 $\implies \exists T' \ rT' \ b.$
 $\quad \text{method } (G, T'') \text{ sig} = \text{Some } (T', rT', b) \wedge$
 $\quad G \vdash rT' \preceq rT \wedge$
 $\quad G \vdash T'' \preceq_C T' \wedge \text{wf_mhead } G \text{ sig } rT' \wedge \text{wf_mb } G \ T' \ (sig, rT', b)$

theorem method_Object:

$\text{method } (\text{tprg}, \text{Object}) = \text{map_of } []$

theorem max_spec2appl_meths:

$x \in \text{max_spec } G \ C \ \text{sig} \implies x \in \text{appl_methds } G \ C \ \text{sig}$

theorem appl_methsD:

$((md, rT), pTs') \in \text{appl_methds } G \ C \ (mn, pTs) \implies$
 $\exists D \ b. \text{md} = \text{Class } D \wedge$
 $\quad \text{method } (G, C) \ (mn, pTs') = \text{Some } (D, rT, b) \wedge$
 $\quad \text{list_all2 } (\lambda T \ T'. G \vdash T \preceq T') \ pTs \ pTs'$

end

References

- [1] G. Klein and T. Nipkow. Verified lightweight bytecode verification. In S. Drossopoulou, S. Eisenbach, B. Jacobs, G. T. Leavens, P. Müller, and A. Poetzsch-Heffter, editors, *Formal Techniques for Java Programs*. Technical Report 269, 5/2000, Fernuniversität Hagen, 2000. ECOOP2000 Workshop proceedings available from <http://www.informatik.fernuni-hagen.de/pi5/publications.html>.
- [2] T. Nipkow, D. v. Oheimb, and C. Pusch. μ Java: 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.
- [3] D. von Oheimb. Axiomatic semantics for Java^{light} 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>.