

LBV for μ Java

Gerwin Klein
Tobias Nipkow

October 11, 2000

Abstract

Eva and Kristoffer Rose proposed a (sparse) annotation of Java Virtual Machine code with types to enable a one-pass verification of welltypedness. We have formalized a variant of their proposal for μ Java in the theorem prover Isabelle/HOL and mechanically verified soundness and completeness.

Contents

1	Lifted Type Relations	2
2	Effect of instructions on the state type	12
3	The Bytecode Verifier	18
4	The Lightweight Bytecode Verifier	20
5	Correctness of the LBV	24
6	Monotonicity of step and app	31
7	Completeness of the LBV	41

1 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* (λ*t* *t*'. *G* ⊢ *t* ≤ *t*')"

sup_loc :: "[code prog, locvars_type, locvars_type] => bool"

("_ ⊢ _ <=l _" [71,71] 70)

"*G* ⊢ *LT* <=l *LT*' == *list_all2* (λ*t* *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 = ( $\exists b. any = Some b \wedge P a b$ )"
  by (simp add: lift_bottom_def split: option.splits)

lemma lift_bottom_Some_any:
  "lift_bottom P any (Some b) = (any = None  $\vee$  ( $\exists a. any = Some a \wedge P a b$ ))"
  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) <=1 (b@y)) = ((G ⊢ a <=1 b) ∧ (G ⊢ x <=1 y))"

proof -

assume l: "length a = length b"

have "∀ b. length a = length b --> (G ⊢ (a@x) <=1 (b@y)) = ((G ⊢ a <=1 b) ∧
(G ⊢ x <=1 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) <=1 (b@y)) = ((G ⊢ (l#ls) <=1 b) ∧ (G ⊢ x <=1 y))"

by - (cases b, auto)

qed

qed

with l

show ?thesis by blast

qed

theorem sup_loc_rev [simp]:

"(G ⊢ (rev a) <=1 rev b) = (G ⊢ a <=1 b)"

proof -

have "∀ b. (G ⊢ (rev a) <=1 rev b) = (G ⊢ a <=1 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) <=1 b"

thus "G ⊢ rev (l # ls) <=1 rev b"

by (clarsimp simp add: Cons IH sup_loc_length sup_loc_append)

next

```

    assume "G ⊢ rev (l # ls) ≤l rev b"
    hence G: "G ⊢ (rev ls @ [l]) ≤l (rev list @ [a])"
      by (simp add: Cons)

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

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

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

  thus ?thesis by blast
qed

theorem sup_loc_update [rule_format]:
  "∀ n y. (G ⊢ a ≤o b) → n < length y → (G ⊢ x ≤l y) →
    (G ⊢ x[n := a] ≤l y[n := b])" (is "?P x")
proof (induct x)
  show "?P []" by simp

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

theorem sup_state_length [simp]:
  "G ⊢ s2 ≤s s1 ==>
  length (fst s2) = length (fst s1) ∧ length (snd s2) = length (snd s1)"
  by (auto dest: sup_loc_length simp add: sup_state_def)

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 ≤₁ b; G ⊢ b ≤₁ c/] ==> G ⊢ a ≤₁ c"
  proof -
    assume G: "G ⊢ a ≤₁ b" "G ⊢ b ≤₁ c"

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

  qed

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

theorem sup_state_trans [trans]:
  "[/G ⊢ a ≤ₛ b; G ⊢ b ≤ₛ c/] ==> G ⊢ a ≤ₛ c"
  by (auto intro: sup_loc_trans simp add: sup_state_def)

theorem sup_state_opt_trans [trans]:
  "[/G ⊢ a ≤' b; G ⊢ b ≤' c/] ==> G ⊢ a ≤' c"
  by (auto intro: lift_bottom_trans sup_state_trans simp add: sup_state_opt_def)

end

```

2 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"

recdef step' "{}"

```

"step' (Load idx, G, (ST, LT))           = (val (LT ! idx) # ST, LT)"
"step' (Store idx, G, (ts#ST, LT))       = (ST, LT[idx:= 0k 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 (λs. step' (i,G,s))"

```

Conditions under which step is applicable:

consts

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

recdef app' "{}"

```

"app' (Load idx, G, rT, s)                = (idx < length (snd s) ∧
  (snd s) ! idx ≠ Err)"
"app' (Store idx, G, rT, (ts#ST, LT))     = (idx < length LT)"
"app' (Bipush i, G, rT, s)                = True"
"app' (Aconst_null, G, rT, s)             = True"
"app' (Getfield F C, G, rT, (oT#ST, LT))  = (is_class G C ∧
  field (G,C) F ≠ None ∧
  fst (the (field (G,C) F)) = C ∧
  G ⊢ oT ≤ (Class C))"
"app' (Putfield F C, G, rT, (vT#oT#ST, LT)) = (is_class G C ∧
  field (G,C) F ≠ None ∧

```

```

fst (the (field (G,C) F)) = C ∧
G ⊢ oT ≲ (Class C) ∧
G ⊢ vT ≲ (snd (the (field (G,C) F)))"
"app' (New C, G, 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)
  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) = ( $\exists apTs\ X\ ST\ LT\ mD'\ rT'\ b'.$ 
 $s = ((rev\ apTs) \textcircled{\#} (X \# ST), LT) \wedge length\ apTs = length\ fpTs \wedge$ 
 $G \vdash X \preceq Class\ C \wedge (\forall (aT,fT) \in set(zip\ apTs\ fpTs). G \vdash aT \preceq fT) \wedge$ 
 $method\ (G,C)\ (mn,fpTs) = Some\ (mD', rT', b')$ )" (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  $\wedge$  ?1")
    by (auto simp add: app_def)
    hence "?a  $\wedge$  0 < length (drop (length fpTs) a)" (is "?a  $\wedge$  ?1")
    by auto
    hence "?a  $\wedge$  ?1  $\wedge$  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

```

3 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

4 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 = 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)))"
  by (auto simp add: wtl_inst_def check_cert_def set_mem_eq)

```

lemma strict_Some [simp]:

```

"(strict f x = Ok y) = (∃ z. x = Ok z ∧ f z = Ok 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 = Ok 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 = 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')"
  (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 Ok: "wtl_cert x G rT s cert mpc pc = Ok s0"

with IH

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

with Ok

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

```

5 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 |]
  ==>  $\forall pc t. pc < length is \rightarrow cert!pc = Some t \rightarrow phi!pc = Some t"$ 

```

```

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  $\neq$  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  $\vdash$  s'' <= phi ! Suc pc"

```

```

proof -

```

```

  assume all: "wtl_inst_list is G rT cert (length is) 0 s  $\neq$  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  $\neq$  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 ]] ==>
 wtl_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. wt_jvm_prog G Phi"
proof (clarsimp simp add: wt_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). wtl_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

```

6 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 ≤₁ y) → n < length y → y!n = Ok t →
(∃ t. b!n = Ok t ∧ (G ⊢ (b!n) ≤₀ (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 ≤₀ 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 ≤₀ Ok t"

proof (cases n)

case 0

with * show ?thesis by (simp add: sup_ty_opt_Ok)

next

case Suc

with Cons *

show ?thesis by (simp add: sup_loc_def)

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) ≤₁ (map Ok b))"

(is "∀ b. length a = length b → ?Q a b" is "?P a")

proof (induct "a")

show "?P []" by simp

fix l ls assume Cons: "?P ls"

show "?P (l#ls)"

proof (intro allI impI)

fix b

assume "length (l # ls) = length (b::ty list)"

with Cons

show "?Q (l # ls) b" by - (cases b, auto)

qed

qed

```

lemma append_length_n [rule_format]:
  "∀n. n ≤ length x --> (∃ a b. x = a@b ∧ length a = n)" (is "?P x")
proof (induct (open) ?P x)
  show "?P []" by simp

  fix l ls assume Cons: "?P ls"

  show "?P (l#ls)"
proof (intro allI impI)
  fix n
  assume l: "n ≤ length (l # ls)"

  show "∃ a b. l # ls = a @ b ∧ length a = n"
proof (cases n)
  assume "n=0" thus ?thesis by simp
next
  fix "n'" assume s: "n = Suc n'"
  with l
  have "n' ≤ length ls" by simp
  hence "∃ a b. ls = a @ b ∧ length a = n'" by (rule Cons [rule_format])
  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 <=' Some 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 <=1 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_Const1)
next
  case New
  with G s step app1 app2
  show ?thesis
    by (clarsimp simp add: sup_state_Const1)
next
  case Aconst_null
  with G s step app1 app2
  show ?thesis
    by (clarsimp simp add: sup_state_Const1)
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 (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

```


7 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' ∈ 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 (succs (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 (succs (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)^*  $\wedge$ 
          ( $\forall (sig, rT, b) \in set ms.$ 
             $\forall D' rT' b'. method (G, D) sig = Some (D', rT', b') \rightarrow G \vdash rT \leq rT'$ ))"
    "(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

```

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.