Java Source and Bytecode Formalizations in Isabelle: μJava

Gerwin Klein Tobias Nipkow David von Oheimb Cornelia Pusch
Martin Strecker

June 21, 2010

Contents

Chapter 1

Preface

1.1 Introduction

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

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

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

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

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

- A “definite assignment” check

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

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

The central topics of the source language formalization are:

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

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

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

- First, the notion of “method type” is introduced, which corresponds to the notion of “type” on the source level.
- Well-typedness of instructions wrt. a method type is defined (see Section 4.17). Roughly speaking, determining well-typedness is *type checking*.
- It is shown that bytecode that is well-typed in this sense can be safely executed – a type soundness proof on the bytecode level (Section 4.21).
- Given raw bytecode, one of the purposes of bytecode verification is to determine a method type that is well-typed according to the above definition. Roughly speaking, this is *type inference*. The Isabelle formalization presents bytecode verification as an instance of an abstract dataflow algorithm (Kildall’s algorithm, see Sections 4.8 to 4.23).

Bytecode verification in μJava so far takes into account:

- Operations and branching instructions
- Exceptions

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

1.2 Theory Dependencies

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

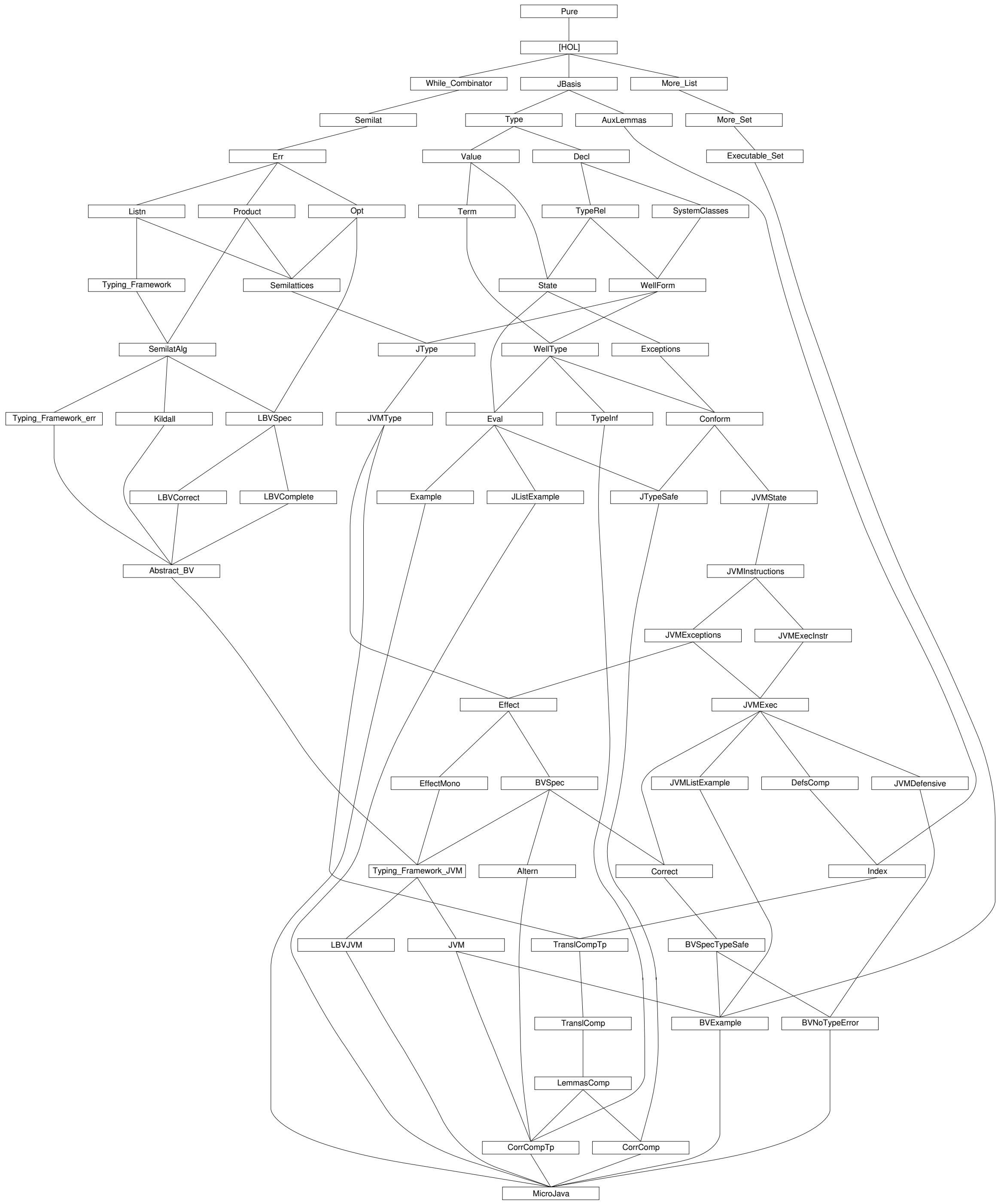

Figure 1.1: Theory Dependency Graph

Chapter 2

Java Source Language

2.1 Some Auxiliary Definitions

theory *JBasis* **imports** *Main* **begin**

lemmas *[simp] = Let_def*

2.1.1 unique

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


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

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

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

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

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

2.1.2 More about Maps

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

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

lemmas *Ball_set_table = Ball_set_table' [THEN mp]*

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

end

2.2 Java types

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

typedecl *cnam*

— exceptions
datatype
```
  xcpt
  = NullPointer
  | ClassCast
  | OutOfMemory
```

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

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

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

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

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

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

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

abbreviation *Class :: "cname => ty"*
  where *"Class C == RefT (ClassT C)"*

end

2.3 Class Declarations and Programs

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

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

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

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

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

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

  'c prog  = "'c cdecl list"       — program
```

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

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

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

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

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

end

2.4 Relations between Java Types

theory *TypeRel* **imports** *Decl* **begin**

— direct subclass, cf. 8.1.3

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

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

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

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

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

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

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

definition class_rec :: "'c prog ⇒ cname ⇒ 'a ⇒
    (cname ⇒ fdecl list ⇒ 'c mdecl list ⇒ 'a ⇒ 'a) ⇒ 'a" where
  "class_rec G == wfrec ((subcls1 G)^-1)
    (λr C t f. case class G C of
```

```
        None ⇒ undefined
      | Some (D,fs,ms) ⇒
          f C fs ms (if C = Object then t else r D t f))"

lemma class_rec_lemma:
  assumes wf: "wf ((subcls1 G)^-1)"
    and cls: "class G C = Some (D, fs, ms)"
  shows "class_rec G C t f = f C fs ms (if C=Object then t else class_rec G D t f)"
proof -
  from wf have step: "⋀H a. wfrec ((subcls1 G)⁻¹) H a =
    H (cut (wfrec ((subcls1 G)⁻¹) H) ((subcls1 G)⁻¹) a) a"
    by (rule wfrec)
  have cut: "⋀f. C ≠ Object ⟹ cut f ((subcls1 G)⁻¹) C D = f D"
    by (rule cut_apply [where r="(subcls1 G)^-1", simplified, OF subcls1I, OF cls])
  from cls show ?thesis by (simp add: step cut class_rec_def)
qed

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

Code generator setup (FIXME!)

```
consts_code
  "wfrec"   ("⟨module⟩wfrec?")
attach {*
fun wfrec f x = f (wfrec f) x;
*}

consts

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

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

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

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

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

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

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

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

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

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

— widening, viz. method invocation conversion,cf. 5.3 i.e. sort of syntactic subtyping

```
inductive
  widen   :: "'c prog => [ty   , ty   ] => bool" ("_ ⊢ _ ≼ _"   [71,71,71] 70)
  for G :: "'c prog"
where
  refl   [intro!, simp]:        "G⊢        T ≼ T"
```

— identity conv., cf. 5.1.1

```
| subcls         : "G⊢C≼C D ==> G⊢Class C ≼ Class D"
| null   [intro!]:              "G⊢     NT ≼ RefT R"
```

```
lemmas refl = HOL.refl
```

— casting conversion, cf. 5.5 / 5.1.5
— left out casts on primitve types

```
inductive
  cast    :: "'c prog => [ty   , ty   ] => bool" ("_ ⊢ _ ≼? _"  [71,71,71] 70)
  for G :: "'c prog"
where
  widen:  "G⊢ C≼ D ==> G⊢C ≼? D"
| subcls: "G⊢ D≼C C ==> G⊢Class C ≼? Class D"
```

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

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

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

```
lemma widen_Class: "G⊢Class C≼T ==> ∃D. T=Class D"
```

```
apply (ind_cases "G⊢Class C≼T")
apply auto
done

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

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

lemma widen_NT_Class [simp]: "G ⊢ T ≼ NT ⟹ G ⊢ T ≼ Class D"
by (ind_cases "G ⊢ T ≼ NT",  auto)

lemma cast_PrimT_RefT [iff]: "(G⊢PrimT pT≼? RefT rT) = False"
apply (rule iffI)
apply (erule cast.cases)
apply auto
done

lemma cast_RefT: "G ⊢ C ≼? Class D ⟹ ∃ rT. C = RefT rT"
apply (erule cast.cases)
apply simp apply (erule widen.cases)
apply auto
done

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

end
```

2.5 Java Values

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

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

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

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

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

primrec
```
  "the_Bool (Bool b) = b"
```

primrec
```
  "the_Intg (Intg i) = i"
```

primrec
```
  "the_Addr (Addr a) = a"
```

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

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

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

end

2.6 Program State

```
theory State
imports TypeRel Value
begin

types
  fields' = "(vname × cname ⇀ val)"   — field name, defining class, value

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

definition obj_ty :: "obj => ty" where
 "obj_ty obj  == Class (fst obj)"

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

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

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

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

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

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

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

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

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

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

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

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

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

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

lemma obj_ty_def2 [simp]: "obj_ty (C,fs) = Class C"
apply (unfold obj_ty_def)
apply (simp (no_asm))
done


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

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

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

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

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

lemma raise_if_SomeD [rule_format (no_asm)]:
  "raise_if c x y = Some z ⟶ c ∧  Some z = Some (Addr (XcptRef x)) |  y = Some z"
apply (unfold raise_if_def)
apply auto
done

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

```
apply auto
done

lemma np_NoneD [rule_format (no_asm)]:
  "np a' x' = None --> x' = None ∧  a' ≠ Null"
apply (unfold np_def raise_if_def)
apply auto
done

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

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

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

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

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

lemma c_hupd_fst [simp]: "fst (c_hupd h (x, s)) = x"
by (simp add: c_hupd_def split_beta)

end
```

2.7 Expressions and Statements

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

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

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

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

end

2.8 System Classes

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

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

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

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

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

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

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

end

2.9 Well-formedness of Java programs

theory *WellForm*
imports *TypeRel SystemClasses*
begin

for static checks on expressions and statements, see WellType.

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

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

simplifications:

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

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

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

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

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

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

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

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

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

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

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

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

lemma wf_cdecl_mrT_cdecl_mdecl:
  "(wf_cdecl wf_mb G c) = (ws_cdecl G c ∧ wf_mrT G c ∧ wf_cdecl_mdecl wf_mb G c)"
apply (rule iffI)
apply (simp add: wf_cdecl_def ws_cdecl_def wf_mrT_def wf_cdecl_mdecl_def
  wf_mdecl_def wf_mhead_def split_beta)+
done

lemma wf_cdecl_ws_cdecl [intro]: "wf_cdecl wf_mb G cd ⟹ ws_cdecl G cd"
by (simp add: wf_cdecl_mrT_cdecl_mdecl)

lemma wf_prog_ws_prog [intro]: "wf_prog wf_mb G ⟹ ws_prog G"
by (simp add: wf_prog_def ws_prog_def)

lemma wf_prog_wf_mdecl:
  "⟦ wf_prog wf_mb G; (C,S,fs,mdecls) ∈ set G; ((mn,pTs),rT,code) ∈ set mdecls⟧
  ⟹ wf_mdecl wf_mb G C ((mn,pTs),rT,code)"
by (auto simp add: wf_prog_def ws_prog_def wf_mdecl_def
  wf_cdecl_mdecl_def ws_cdecl_def)

lemma class_wf:
 "[|class G C = Some c; wf_prog wf_mb G|]
  ==> wf_cdecl wf_mb G (C,c) ∧ wf_mrT G (C,c)"
apply (unfold wf_prog_def ws_prog_def wf_cdecl_def class_def)
apply clarify
apply (drule_tac x="(C,c)" in bspec, fast dest: map_of_SomeD)
apply (drule_tac x="(C,c)" in bspec, fast dest: map_of_SomeD)
apply (simp add: wf_cdecl_def ws_cdecl_def wf_mdecl_def
  wf_cdecl_mdecl_def wf_mrT_def split_beta)
done

lemma class_wf_struct:
 "[|class G C = Some c; ws_prog G|]
  ==> ws_cdecl G (C,c)"
apply (unfold ws_prog_def class_def)
apply (fast dest: map_of_SomeD)
done

lemma class_Object [simp]:
  "ws_prog G ==> ∃X fs ms. class G Object = Some (X,fs,ms)"
apply (unfold ws_prog_def wf_syscls_def class_def)
apply (auto simp: map_of_SomeI)
```

```
done

lemma class_Object_syscls [simp]:
  "wf_syscls G ==> unique G ⟹ ∃X fs ms. class G Object = Some (X,fs,ms)"
apply (unfold wf_syscls_def class_def)
apply (auto simp: map_of_SomeI)
done

lemma is_class_Object [simp]: "ws_prog G ==> is_class G Object"
  by (simp add: is_class_def)

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

lemma subcls1_wfD: "[|G⊢C≺C1D; ws_prog G|] ==> D ≠ C ∧ (D, C) ∉ (subcls1 G)^+"
apply( frule trancl.r_into_trancl [where r="subcls1 G"])
apply( drule subcls1D)
apply(clarify)
apply( drule (1) class_wf_struct)
apply( unfold ws_cdecl_def)
apply(force simp add: reflcl_trancl [THEN sym] simp del: reflcl_trancl)
done

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

lemma subcls_asym: "[|ws_prog G; (C, D) ∈ (subcls1 G)^+|] ==> (D, C) ∉ (subcls1 G)^+"
apply(erule trancl.cases)
apply(fast dest!: subcls1_wfD )
apply(fast dest!: subcls1_wfD intro: trancl_trans)
done

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

lemma acyclic_subcls1: "ws_prog G ==> acyclic (subcls1 G)"
apply (simp add: acyclic_def)
apply (fast dest: subcls_irrefl)
done

lemma wf_subcls1: "ws_prog G ==> wf ((subcls1 G)^-1)"
apply (rule finite_acyclic_wf)
apply ( subst finite_converse)
```

```
apply ( rule finite_subcls1)
apply (subst acyclic_converse)
apply (erule acyclic_subcls1)
done

lemma subcls_induct_struct:
"[|ws_prog G; !!C. ∀D. (C, D) ∈ (subcls1 G)^+ --> P D ==> P C|] ==> P C"
(is "?A ⟹ PROP ?P ⟹ _")
proof -
  assume p: "PROP ?P"
  assume ?A thus ?thesis apply -
apply(drule wf_subcls1)
apply(drule wf_trancl)
apply(simp only: trancl_converse)
apply(erule_tac a = C in wf_induct)
apply(rule p)
apply(auto)
done
qed

lemma subcls_induct:
"[|wf_prog wf_mb G; !!C. ∀D. (C, D) ∈ (subcls1 G)^+ --> P D ==> P C|] ==> P C"
(is "?A ⟹ PROP ?P ⟹ _")
by (fact subcls_induct_struct [OF wf_prog_ws_prog])

lemma subcls1_induct:
"[|is_class G C; wf_prog wf_mb G; P Object;
   !!C D fs ms. [|C ≠ Object; is_class G C; class G C = Some (D,fs,ms) ∧
    wf_cdecl wf_mb G (C,D,fs,ms) ∧ G⊢C≺C1D ∧ is_class G D ∧ P D|] ==> P C
 |] ==> P C"
(is "?A ⟹ ?B ⟹ ?C ⟹ PROP ?P ⟹ _")
proof -
  assume p: "PROP ?P"
  assume ?A ?B ?C thus ?thesis apply -
apply(unfold is_class_def)
apply( rule impE)
prefer 2
apply(   assumption)
prefer 2
apply(  assumption)
apply( erule thin_rl)
apply( rule subcls_induct)
apply(  assumption)
apply( rule impI)
apply( case_tac "C = Object")
apply(  fast)
apply auto
apply( frule (1) class_wf) apply (erule conjE)+
apply (frule wf_cdecl_ws_cdecl)
apply( frule (1) wf_cdecl_supD)

apply( subgoal_tac "G⊢C≺C1a")
apply( erule_tac [2] subcls1I)
apply(  rule p)
```

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

lemma subcls1_induct_struct:
"[|is_class G C; ws_prog G; P Object;
   !!C D fs ms. [|C ≠ Object; is_class G C; class G C = Some (D,fs,ms) ∧
    ws_cdecl G (C,D,fs,ms) ∧ G⊢C≺C1D ∧ is_class G D ∧ P D|] ==> P C
 |] ==> P C"
(is "?A ⟹ ?B ⟹ ?C ⟹ PROP ?P ⟹ _")
proof -
  assume p: "PROP ?P"
  assume ?A ?B ?C thus ?thesis apply -
apply(unfold is_class_def)
apply( rule impE)
prefer 2
apply(   assumption)
prefer 2
apply(  assumption)
apply( erule thin_rl)
apply( rule subcls_induct_struct)
apply(  assumption)
apply( rule impI)
apply( case_tac "C = Object")
apply(  fast)
apply auto
apply( frule (1) class_wf_struct)
apply( frule (1) wf_cdecl_supD)

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

lemmas method_rec = wf_subcls1 [THEN [2] method_rec_lemma]

lemmas fields_rec = wf_subcls1 [THEN [2] fields_rec_lemma]

lemma field_rec: "⟦class G C = Some (D, fs, ms); ws_prog G⟧
⟹ field (G, C) =
   (if C = Object then empty else field (G, D)) ++
   map_of (map (λ(s, f). (s, C, f)) fs)"
apply (simp only: field_def)
apply (frule fields_rec, assumption)
apply (rule HOL.trans)
apply (simp add: o_def)
apply (simp (no_asm_use) add: split_beta split_def o_def)
done

lemma method_Object [simp]:
```

```
  "method (G, Object) sig = Some (D, mh, code) ⟹ ws_prog G ⟹ D = Object"
  apply (frule class_Object, clarify)
  apply (drule method_rec, assumption)
  apply (auto dest: map_of_SomeD)
  done


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

lemma subcls_C_Object: "[|is_class G C; ws_prog G|] ==> G⊢C≼C Object"
apply(erule subcls1_induct_struct)
apply(  assumption)
apply( fast)
apply(auto dest!: wf_cdecl_supD)
done

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

lemma widen_fields_defpl': "[|is_class G C; ws_prog G|] ==>
  ∀((fn,fd),fT)∈set (fields (G,C)). G⊢C≼C fd"
apply( erule subcls1_induct_struct)
apply(   assumption)
apply(  frule class_Object)
apply(  clarify)
apply(  frule fields_rec, assumption)
apply(  fastsimp)
apply( tactic "safe_tac HOL_cs")
apply( subst fields_rec)
apply(   assumption)
apply(  assumption)
apply( simp (no_asm) split del: split_if)
apply( rule ballI)
apply( simp (no_asm_simp) only: split_tupled_all)
apply( simp (no_asm))
apply( erule UnE)
apply(  force)
apply( erule r_into_rtrancl [THEN rtrancl_trans])
apply auto
done

lemma widen_fields_defpl:
```

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

lemma unique_fields:
  "[|is_class G C; ws_prog G|] ==> unique (fields (G,C))"
apply( erule subcls1_induct_struct)
apply(   assumption)
apply(  frule class_Object)
apply(  clarify)
apply(  frule fields_rec, assumption)
apply(  drule class_wf_struct, assumption)
apply(  simp add: ws_cdecl_def)
apply(  rule unique_map_inj)
apply(   simp)
apply(  rule inj_onI)
apply(  simp)
apply( safe dest!: wf_cdecl_supD)
apply( drule subcls1_wfD)
apply(  assumption)
apply( subst fields_rec)
apply   auto
apply( rotate_tac -1)
apply( frule class_wf_struct)
apply  auto
apply( simp add: ws_cdecl_def)
apply( erule unique_append)
apply(  rule unique_map_inj)
apply(   clarsimp)
apply  (rule inj_onI)
apply(  simp)
apply(auto dest!: widen_fields_defpl)
done

lemma fields_mono_lemma [rule_format (no_asm)]:
  "[|ws_prog G; (C', C) ∈ (subcls1 G)^*|] ==>
  x ∈ set (fields (G,C)) --> x ∈ set (fields (G,C'))"
apply(erule converse_rtrancl_induct)
apply( safe dest!: subcls1D)
apply(subst fields_rec)
apply(  auto)
done

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

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

lemma method_wf_mdecl [rule_format (no_asm)]:
  "wf_prog wf_mb G ==> is_class G C ⟹
     method (G,C) sig = Some (md,mh,m)
   --> G⊢C≼C md ∧ wf_mdecl wf_mb G md (sig,(mh,m))"
apply (frule wf_prog_ws_prog)
apply( erule subcls1_induct)
apply(   assumption)
apply(  clarify)
apply(  frule class_Object)
apply(  clarify)
apply(  frule method_rec, assumption)
apply(  drule class_wf, assumption)
apply(  simp add: wf_cdecl_def)
apply(  drule map_of_SomeD)
apply(  subgoal_tac "md = Object")
apply(   fastsimp)
apply(  fastsimp)
apply( clarify)
apply( frule_tac C = C in method_rec)
apply(  assumption)
apply( rotate_tac -1)
apply( simp)
apply( drule map_add_SomeD)
apply( erule disjE)
apply(  erule_tac V = "?P --> ?Q" in thin_rl)
apply (frule map_of_SomeD)
apply (clarsimp simp add: wf_cdecl_def)
apply( clarify)
apply( rule rtrancl_trans)
prefer 2
apply(  assumption)
apply( rule r_into_rtrancl)
apply( fast intro: subcls1I)
done


lemma method_wf_mhead [rule_format (no_asm)]:
  "ws_prog G ==> is_class G C ⟹
     method (G,C) sig = Some (md,rT,mb)
   --> G⊢C≼C md ∧ wf_mhead G sig rT"
apply( erule subcls1_induct_struct)
apply(   assumption)
apply(  clarify)
```

```
apply(  frule class_Object)
apply(  clarify)
apply(  frule method_rec, assumption)
apply(  drule class_wf_struct, assumption)
apply(  simp add: ws_cdecl_def)
apply(  drule map_of_SomeD)
apply(  subgoal_tac "md = Object")
apply(   fastsimp)
apply(  fastsimp)
apply( clarify)
apply( frule_tac C = C in method_rec)
apply(  assumption)
apply( rotate_tac -1)
apply( simp)
apply( drule map_add_SomeD)
apply( erule disjE)
apply(  erule_tac V = "?P --> ?Q" in thin_rl)
apply (frule map_of_SomeD)
apply (clarsimp simp add: ws_cdecl_def)
apply blast
apply clarify
apply( rule rtrancl_trans)
prefer 2
apply(  assumption)
apply( rule r_into_rtrancl)
apply( fast intro: subcls1I)
done

lemma subcls_widen_methd [rule_format (no_asm)]:
  "[|G⊢T'≼C T; wf_prog wf_mb G|] ==>
   ∀D rT b. method (G,T) sig = Some (D,rT ,b) -->
  (∃D' rT' b'. method (G,T') sig = Some (D',rT',b') ∧ G⊢D'≼C D ∧ G⊢rT'≼rT)"
apply( drule rtranclD)
apply( erule disjE)
apply(  fast)
apply( erule conjE)
apply( erule trancl_trans_induct)
prefer 2
apply(  clarify)
apply(  drule spec, drule spec, drule spec, erule (1) impE)
apply(  fast elim: widen_trans rtrancl_trans)
apply( clarify)
apply( drule subcls1D)
apply( clarify)
apply( subst method_rec)
apply(  assumption)
apply( unfold map_add_def)
apply( simp (no_asm_simp) add: wf_prog_ws_prog del: split_paired_Ex)
apply( case_tac "∃z. map_of(map (λ(s,m). (s, ?C, m)) ms) sig = Some z")
apply(  erule exE)
apply(  rotate_tac -1, frule ssubst, erule_tac [2] asm_rl)
prefer 2
apply(  rotate_tac -1, frule ssubst, erule_tac [2] asm_rl)
apply(  tactic "asm_full_simp_tac (HOL_ss addsimps [@{thm not_None_eq} RS sym]) 1")
```

```
apply(  simp_all (no_asm_simp) del: split_paired_Ex)
apply( frule (1) class_wf)
apply( simp (no_asm_simp) only: split_tupled_all)
apply( unfold wf_cdecl_def)
apply( drule map_of_SomeD)
apply (auto simp add: wf_mrT_def)
apply (rule rtrancl_trans)
defer
apply (rule method_wf_mhead [THEN conjunct1])
apply (simp only: wf_prog_def)
apply (simp add: is_class_def)
apply assumption
apply (auto intro: subcls1I)
done
```

```
lemma subtype_widen_methd:
 "[| G⊢ C≼C D; wf_prog wf_mb G;
     method (G,D) sig = Some (md, rT, b) |]
  ==> ∃mD' rT' b'. method (G,C) sig= Some(mD',rT',b') ∧ G⊢rT'≼rT"
apply(auto dest: subcls_widen_methd
           simp add: wf_mdecl_def wf_mhead_def split_def)
done
```

```
lemma method_in_md [rule_format (no_asm)]:
  "ws_prog G ==> is_class G C ⟹ ∀D. method (G,C) sig = Some(D,mh,code)
  --> is_class G D ∧ method (G,D) sig = Some(D,mh,code)"
apply (erule (1) subcls1_induct_struct)
 apply clarify
 apply (frule method_Object, assumption)
 apply hypsubst
 apply simp
apply (erule conjE)
apply (simplesubst method_rec, assumption+)
apply (clarify)
apply (erule_tac x = "Da" in allE)
apply (clarsimp)
 apply (simp add: map_of_map)
 apply (clarify)
 apply (subst method_rec, assumption+)
 apply (simp add: map_add_def map_of_map split add: option.split)
done
```

```
lemma method_in_md_struct [rule_format (no_asm)]:
  "ws_prog G ==> is_class G C ⟹ ∀D. method (G,C) sig = Some(D,mh,code)
  --> is_class G D ∧ method (G,D) sig = Some(D,mh,code)"
apply (erule (1) subcls1_induct_struct)
 apply clarify
 apply (frule method_Object, assumption)
 apply hypsubst
 apply simp
apply (erule conjE)
```

```
apply (simplesubst method_rec, assumption+)
apply (clarify)
apply (erule_tac x = "Da" in allE)
apply (clarsimp)
 apply (simp add: map_of_map)
 apply (clarify)
 apply (subst method_rec, assumption+)
 apply (simp add: map_add_def map_of_map split add: option.split)
done

lemma fields_in_fd [rule_format (no_asm)]: "⟦ wf_prog wf_mb G; is_class G C⟧
  ⟹ ∀ vn D T. (((vn,D),T) ∈ set (fields (G,C))
  ⟶ (is_class G D ∧ ((vn,D),T) ∈ set (fields (G,D))))"
apply (erule (1) subcls1_induct)

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

apply clarify
apply (frule fields_rec)
apply (simp (no_asm_simp) add: wf_prog_ws_prog)

apply (case_tac "Da=C")
apply blast

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

lemma field_in_fd [rule_format (no_asm)]: "⟦ wf_prog wf_mb G; is_class G C⟧
  ⟹ ∀ vn D T. (field (G,C) vn = Some(D,T)
  ⟶ is_class G D ∧ field (G,D) vn = Some(D,T))"
apply (erule (1) subcls1_induct)

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

apply clarify
apply (subgoal_tac "((field (G, D)) ++ map_of (map (λ(s, f). (s, C, f)) fs)) vn = Some
(Da, T)")
apply (simp (no_asm_use) only: map_add_Some_iff)
apply (erule disjE)
apply (simp (no_asm_use) add: map_of_map) apply blast
apply blast
```

```
apply (rule trans [THEN sym], rule sym, assumption)
apply (rule_tac x=vn in fun_cong)
apply (rule trans, rule field_rec, assumption+)
apply (simp (no_asm_simp) add: wf_prog_ws_prog)
apply (simp (no_asm_use)) apply blast
done


lemma widen_methd:
"[| method (G,C) sig = Some (md,rT,b); wf_prog wf_mb G; G⊢T''≼C C|]
  ==> ∃md' rT' b'. method (G,T'') sig = Some (md',rT',b') ∧ G⊢rT'≼rT"
apply( drule subcls_widen_methd)
apply   auto
done

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

lemma Call_lemma:
"[|method (G,C) sig = Some (md,rT,b); G⊢T''≼C C; wf_prog wf_mb G;
  class G C = Some y|] ==> ∃T' rT' b. method (G,T'') sig = Some (T',rT',b) ∧
  G⊢rT'≼rT ∧ G⊢T''≼C T' ∧ wf_mhead G sig rT' ∧ wf_mb G T' (sig,rT',b)"
apply( drule (2) widen_methd)
apply( clarify)
apply( frule subcls_is_class2)
apply (unfold is_class_def)
apply (simp (no_asm_simp))
apply( drule method_wf_mdecl)
apply( unfold wf_mdecl_def)
apply( unfold is_class_def)
apply auto
done

lemma fields_is_type_lemma [rule_format (no_asm)]:
  "[|is_class G C; ws_prog G|] ==>
  ∀f∈set (fields (G,C)). is_type G (snd f)"
apply( erule (1) subcls1_induct_struct)
apply(  frule class_Object)
apply(  clarify)
apply(  frule fields_rec, assumption)
apply(  drule class_wf_struct, assumption)
apply(  simp add: ws_cdecl_def wf_fdecl_def)
apply(  fastsimp)
apply( subst fields_rec)
apply(   fast)
apply(  assumption)
```

```
apply( clarsimp)
apply( safe)
prefer 2
apply(  force)
apply( drule (1) class_wf_struct)
apply( unfold ws_cdecl_def)
apply( clarsimp)
apply( drule (1) bspec)
apply( unfold wf_fdecl_def)
apply auto
done


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


lemma field_is_type: "⟦ ws_prog G; is_class G C; field (G, C) fn = Some (fd, fT) ⟧
  ⟹ is_type G fT"
apply (frule_tac f="((fn, fd), fT)" in fields_is_type_lemma)
apply (auto simp add: field_def dest: map_of_SomeD)
done


lemma methd:
  "[| ws_prog G; (C,S,fs,mdecls) ∈ set G; (sig,rT,code) ∈ set mdecls |]
  ==> method (G,C) sig = Some(C,rT,code) ∧ is_class G C"
proof -
  assume wf: "ws_prog G" and C:  "(C,S,fs,mdecls) ∈ set G" and
         m: "(sig,rT,code) ∈ set mdecls"
  moreover
  from wf C have "class G C = Some (S,fs,mdecls)"
    by (auto simp add: ws_prog_def class_def is_class_def intro: map_of_SomeI)
  moreover
  from wf C
  have "unique mdecls" by (unfold ws_prog_def ws_cdecl_def) auto
  hence "unique (map (λ(s,m). (s,C,m)) mdecls)" by (induct mdecls, auto)
  with m
  have "map_of (map (λ(s,m). (s,C,m)) mdecls) sig = Some (C,rT,code)"
    by (force intro: map_of_SomeI)
  ultimately
  show ?thesis by (auto simp add: is_class_def dest: method_rec)
qed


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

```
  apply (simp only: wf_prog_def)
  apply auto
  apply (simp add: wf_cdecl_mdecl_def)
  apply safe
  apply (drule bspec, assumption) apply simp
  done


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

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

end
```

2.10 Well-typedness Constraints

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

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

simplifications:

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

local variables, including method parameters and This:

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

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

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

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

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

  — applicable methods, cf. 15.11.2.1
  appl_methds_def: "appl_methds G C == λ(mn, pTs).
                     {((Class md,rT),pTs') |md rT mb pTs'.
                      method (G,C)  (mn, pTs') = Some (md,rT,mb) ∧
                      list_all2 (λT T'. G⊢T≼T') pTs pTs'}"

  — maximally specific methods, cf. 15.11.2.2
  max_spec_def: "max_spec G C sig == {m. m ∈appl_methds G C sig ∧
                                      (∀m'∈appl_methds G C sig.
                                        more_spec G m' m --> m' = m)}"

lemma max_spec2appl_meths:
  "x ∈ max_spec G C sig ==> x ∈ appl_methds G C sig"
apply (unfold max_spec_def)
apply (fast)
done
```

```
lemma appl_methsD:
"((md,rT),pTs')∈appl_methds G C (mn, pTs) ==>
  ∃D b. md = Class D ∧ method (G,C) (mn, pTs') = Some (D,rT,b)
  ∧ list_all2 (λT T'. G⊢T≼T') pTs pTs'"
apply (unfold appl_methds_def)
apply (fast)
done

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


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

primrec
  "typeof dt  Unit    = Some (PrimT Void)"
  "typeof dt  Null    = Some NT"
  "typeof dt (Bool b) = Some (PrimT Boolean)"
  "typeof dt (Intg i) = Some (PrimT Integer)"
  "typeof dt (Addr a) = dt a"

lemma is_type_typeof [rule_format (no_asm), simp]:
  "(∀a. v ≠ Addr a) --> (∃T. typeof t v = Some T ∧ is_type G T)"
apply (rule val.induct)
apply     auto
done

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

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

types
  java_mb = "vname list × (vname × ty) list × stmt × expr"
```

— method body with parameter names, local variables, block, result expression.
— local variables might include This, which is hidden anyway

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

  NewC: "[| is_class (prg E) C |] ==>
```

```
        E⊢NewC C::Class C"
```
— cf. 15.8

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

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

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

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

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

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

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

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

— well-typed expression lists

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

— cf. 15.11.???
```
| Cons:"[| E⊢e::T;
```

```
            E⊢es[::]Ts |] ==>
         E⊢e#es[::]T#Ts"
```

— well-typed statements

```
| Skip:"E⊢Skip√"

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

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

— cf. 14.8

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

— cf. 14.10

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

definition *wf_java_mdecl :: "'c prog => cname => java_mb mdecl => bool"* **where**

```
"wf_java_mdecl G C == λ((mn,pTs),rT,(pns,lvars,blk,res)).
  length pTs = length pns ∧
  distinct pns ∧
  unique lvars ∧
        This ∉ set pns ∧ This ∉ set (map fst lvars) ∧
  (∀pn∈set pns. map_of lvars pn = None) ∧
  (∀(vn,T)∈set lvars. is_type G T) &
  (let E = (G,map_of lvars(pns[↦]pTs)(This↦Class C)) in
   E⊢blk√ ∧ (∃T. E⊢res::T ∧ G⊢T≼rT))"
```

abbreviation *"wf_java_prog == wf_prog wf_java_mdecl"*

lemma *wf_java_prog_wf_java_mdecl: "⟦*
wf_java_prog G; (C, D, fds, mths) ∈ set G; jmdcl ∈ set mths ⟧
⟹ wf_java_mdecl G C jmdcl"
apply *(simp only: wf_prog_def)*
apply *(erule conjE)+*
apply *(drule bspec, assumption)*
apply *(simp add: wf_cdecl_mdecl_def split_beta)*
done

lemma *wt_is_type: "(E⊢e::T ⟶ ws_prog (prg E) ⟶ is_type (prg E) T) ∧*
(E⊢es[::]Ts ⟶ ws_prog (prg E) ⟶ Ball (set Ts) (is_type (prg E))) ∧
(E⊢c √ ⟶ True)"
apply *(rule ty_expr_ty_exprs_wt_stmt.induct)*
apply *auto*

```
apply (   erule typeof_empty_is_type)
apply (  simp split add: split_if_asm)
apply ( drule field_fields)
apply ( drule (1) fields_is_type)
apply (  simp (no_asm_simp))
apply  (assumption)
apply (auto dest!: max_spec2mheads method_wf_mhead is_type_rTI
           simp add: wf_mdecl_def)
done

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

lemma expr_class_is_class: "
  ⟦ws_prog (prg E); E ⊢ e :: Class C⟧ ⟹ is_class (prg E) C"
  by (frule ty_expr_is_type, assumption, simp)


end
```

2.11 Operational Evaluation (big step) Semantics

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

— Auxiliary notions

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

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

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

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

— Evaluation relations

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

— evaluation of expressions

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

— cf. 15.8.1

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

— cf. 15.15

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

— cf. 15.7.1

```
| Lit:  "G⊢Norm s -Lit v≻v-> Norm s"

| BinOp:"[| G⊢Norm s -e1≻v1-> s1;
```

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

— cf. 15.13.1, 15.2

```
| LAcc: "G⊢Norm s -LAcc v≻the (locals s v)-> Norm s"
```

— cf. 15.25.1

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

— cf. 15.10.1, 15.2

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

— cf. 15.25.1

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

— cf. 15.11.4.1, 15.11.4.2, 15.11.4.4, 15.11.4.5, 14.15

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

— evaluation of expression lists

— cf. 15.5

```
| XcptEs:"G⊢(Some xc,s) -e[≻]undefined-> (Some xc,s)"
```

— cf. 15.11.???

```
| Nil:  "G⊢Norm s0 -[][≻][]-> Norm s0"
```

— cf. 15.6.4

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

— execution of statements

— cf. 14.1

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

— cf. 14.5

```
| Skip: "G⊢Norm s -Skip-> Norm s"
```

— cf. 14.7

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

— cf. 14.2

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

— cf. 14.8.2

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

— cf. 14.10, 14.10.1

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

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

lemma NewCI: "[|new_Addr (heap s) = (a,x);
       s' = c_hupd (heap s(a↦(C,init_vars (fields (G,C))))) (x,s)|] ==>
       G⊢Norm s -NewC C≻Addr a-> s'"
apply (simp (no_asm_simp))
apply (rule eval_evals_exec.NewC)
apply auto
done

lemma eval_evals_exec_no_xcpt:
 "!!s s'. (G⊢(x,s) -e ≻  v -> (x',s') --> x'=None --> x=None) ∧
          (G⊢(x,s) -es[≻]vs-> (x',s') --> x'=None --> x=None) ∧
          (G⊢(x,s) -c       -> (x',s') --> x'=None --> x=None)"
apply(simp (no_asm_simp) only: split_tupled_all)
apply(rule eval_evals_exec_induct)
apply(unfold c_hupd_def)
apply(simp_all)
done

lemma eval_no_xcpt: "G⊢(x,s) -e≻v-> (None,s') ==> x=None"
apply (drule eval_evals_exec_no_xcpt [THEN conjunct1, THEN mp])
apply (fast)
done

lemma evals_no_xcpt: "G⊢(x,s) -e[≻]v-> (None,s') ==> x=None"
apply (drule eval_evals_exec_no_xcpt [THEN conjunct2, THEN conjunct1, THEN mp])
apply (fast)
done
```

```
lemma exec_no_xcpt: "G ⊢ (x, s) -c-> (None, s')
⟹ x = None"
apply (drule eval_evals_exec_no_xcpt [THEN conjunct2 [THEN conjunct2], rule_format])
apply simp+
done
```

```
lemma eval_evals_exec_xcpt:
"!!s s'. (G⊢(x,s) -e ≻  v -> (x',s') --> x=Some xc --> x'=Some xc ∧ s'=s) ∧
         (G⊢(x,s) -es[≻]vs-> (x',s') --> x=Some xc --> x'=Some xc ∧ s'=s) ∧
         (G⊢(x,s) -c       -> (x',s') --> x=Some xc --> x'=Some xc ∧ s'=s)"
apply (simp (no_asm_simp) only: split_tupled_all)
apply (rule eval_evals_exec_induct)
apply (unfold c_hupd_def)
apply (simp_all)
done
```

```
lemma eval_xcpt: "G⊢(Some xc,s) -e≻v-> (x',s') ==> x'=Some xc ∧  s'=s"
apply (drule eval_evals_exec_xcpt [THEN conjunct1, THEN mp])
apply (fast)
done
```

```
lemma exec_xcpt: "G⊢(Some xc,s) -s0-> (x',s') ==> x'=Some xc ∧  s'=s"
apply (drule eval_evals_exec_xcpt [THEN conjunct2, THEN conjunct2, THEN mp])
apply (fast)
done
```

```
end
```

```
theory Exceptions imports State begin
```

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

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

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

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

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

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

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

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

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


end
```

2.12 Conformity Relations for Type Soundness Proof

```
theory Conform imports State WellType Exceptions begin

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

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

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

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

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

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

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

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


notation (xsymbols)
  hext  ("_ ≤| _" [51,51] 50) and
  conf  ("_,_ ⊢ _ ::≼ _" [51,51,51,51] 50) and
  lconf  ("_,_ ⊢ _ [::≼] _" [51,51,51,51] 50) and
  oconf  ("_,_ ⊢ _ √" [51,51,51] 50) and
  hconf  ("_ ⊢h _ √" [51,51] 50) and
  conforms  ("_ ::≼ _" [51,51] 50)
```

2.12.1 hext

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

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

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

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

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

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

2.12.2 conf

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

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

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

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

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

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

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

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

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

lemma conf_RefTD [rule_format (no_asm)]:
 "G,h⊢a'::≼RefT T --> a' = Null |
  (∃a obj T'. a' = Addr a ∧  h a = Some obj ∧  obj_ty obj = T' ∧  G⊢T'≼RefT T)"
apply (unfold conf_def)
apply(induct_tac "a'")
apply(auto)
done

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

lemma non_npD: "[|a' ≠ Null; G,h⊢a'::≼RefT t|] ==>
  ∃a C fs. a' = Addr a ∧  h a = Some (C,fs) ∧  G⊢Class C≼RefT t"
apply (drule conf_RefTD)
apply auto
done

lemma non_np_objD: "!!G. [|a' ≠ Null; G,h⊢a'::≼ Class C|] ==>
  (∃a C' fs. a' = Addr a ∧  h a = Some (C',fs) ∧  G⊢C'≼C C)"
apply (fast dest: non_npD)
done

lemma non_np_objD' [rule_format (no_asm)]:
  "a' ≠ Null ==> wf_prog wf_mb G ==> G,h⊢a'::≼RefT t -->
  (∃a C fs. a' = Addr a ∧  h a = Some (C,fs) ∧  G⊢Class C≼RefT t)"
apply(rule_tac y = "t" in ref_ty.exhaust)
 apply (fast dest: conf_NullTD)
```

```
apply (fast dest: non_np_objD)
done

lemma conf_list_gext_widen [rule_format (no_asm)]:
  "wf_prog wf_mb G ==> ∀Ts Ts'. list_all2 (conf G h) vs Ts -->
  list_all2 (λT T'. G⊢T≼T') Ts Ts' -->  list_all2 (conf G h) vs Ts'"
apply(induct_tac "vs")
 apply(clarsimp)
apply(clarsimp)
apply(frule list_all2_lengthD [THEN sym])
apply(simp (no_asm_use) add: length_Suc_conv)
apply(safe)
apply(frule list_all2_lengthD [THEN sym])
apply(simp (no_asm_use) add: length_Suc_conv)
apply(clarify)
apply(fast elim: conf_widen)
done
```

2.12.3 lconf

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

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

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

lemma lconf_init_vars_lemma [rule_format (no_asm)]:
  "∀x. P x --> R (dv x) x ==> (∀x. map_of fs f = Some x --> P x) -->
  (∀T. map_of fs f = Some T -->
  (∃v. map_of (map (λ(f,ft). (f, dv ft)) fs) f = Some v ∧  R v T))"
apply( induct_tac "fs")
apply auto
done

lemma lconf_init_vars [intro!]:
"∀n. ∀T. map_of fs n = Some T --> is_type G T ==> G,h⊢init_vars fs[::≼]map_of fs"
apply (unfold lconf_def init_vars_def)
apply auto
apply( rule lconf_init_vars_lemma)
apply(   erule_tac [3] asm_rl)
apply(  intro strip)
apply(  erule defval_conf)
apply auto
done
```

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

lemma lconf_ext_list [rule_format (no_asm)]:
  "G,h⊢l[::≼]L ==> ∀vs Ts. distinct vns --> length Ts = length vns -->
  list_all2 (λv T. G,h⊢v::≼T) vs Ts --> G,h⊢l(vns[↦]vs)[::≼]L(vns[↦]Ts)"
apply (unfold lconf_def)
apply( induct_tac "vns")
apply(  clarsimp)
apply( clarsimp)
apply( frule list_all2_lengthD)
apply( auto simp add: length_Suc_conv)
done

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

2.12.4 oconf

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

lemma oconf_obj: "G,h⊢(C,fs)√ =
  (∀T f. map_of(fields (G,C)) f = Some T --> (∃v. fs f = Some v ∧  G,h⊢v::≼T))"
apply (unfold oconf_def lconf_def)
apply auto
done

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

2.12.5 hconf

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

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

2.12.6 xconf

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

2.12.7 conforms

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

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

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

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

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

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


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

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

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


lemma conforms_upd_obj:
  "[|(x,(h,l))::≼(G, lT); G,h(a↦obj)⊢obj√; h≤|h(a↦obj)|]
  ==> (x,(h(a↦obj),l))::≼(G, lT)"
apply(rule conforms_hext)
apply  auto
apply(rule hconfI)
apply(drule conforms_heapD)
apply(tactic {* auto_tac (HOL_cs addEs [@{thm oconf_hext}]
                addDs [@{thm hconfD}], @{simpset} delsimps [@{thm split_paired_All}])
*})
done

lemma conforms_upd_local:
```

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

end

2.13 Type Safety Proof

```
theory JTypeSafe imports Eval Conform begin

declare split_beta [simp]

lemma NewC_conforms:
"[|h a = None; (x,(h, l))::≼(G, lT); wf_prog wf_mb G; is_class G C|] ==>
  (x,(h(a↦(C,(init_vars (fields (G,C))))), l))::≼(G, lT)"
apply( erule conforms_upd_obj)
apply(  unfold oconf_def)
apply(  auto dest!: fields_is_type simp add: wf_prog_ws_prog)
done


lemma Cast_conf:
 "[| wf_prog wf_mb G; G,h⊢v::≼CC; G⊢CC ≼? Class D; cast_ok G D h v|]
  ==> G,h⊢v::≼Class D"
apply (case_tac "CC")
apply simp
apply (case_tac "ref_ty")
apply (clarsimp simp add: conf_def)
apply simp
apply (ind_cases "G ⊢ Class cname ≼? Class D" for cname, simp)
apply (rule conf_widen, assumption+) apply (erule widen.subcls)

apply (unfold cast_ok_def)
apply( case_tac "v = Null")
apply(  simp)
apply(  clarify)
apply( drule (1) non_npD)
apply( auto intro!: conf_AddrI simp add: obj_ty_def)
done


lemma FAcc_type_sound:
"[| wf_prog wf_mb G; field (G,C) fn = Some (fd, ft); (x,(h,l))::≼(G,lT);
  x' = None --> G,h⊢a'::≼ Class C; np a' x' = None |] ==>
  G,h⊢the (snd (the (h (the_Addr a'))) (fn, fd))::≼ft"
apply( drule np_NoneD)
apply( erule conjE)
apply( erule (1) notE impE)
apply( drule non_np_objD)
apply   auto
apply( drule conforms_heapD [THEN hconfD])
apply(  assumption)
apply (frule wf_prog_ws_prog)
apply( drule (2) widen_cfs_fields)
apply( drule (1) oconf_objD)
apply auto
done

lemma FAss_type_sound:
 "[| wf_prog wf_mb G; a = the_Addr a'; (c, fs) = the (h a);
```

```
    (G, lT)⊢v::T'; G⊢T'≼ft;
    (G, lT)⊢aa::Class C;
    field (G,C) fn = Some (fd, ft); h''≤|h';
    x' = None --> G,h'⊢a'::≼ Class C; h'≤|h;
    Norm (h, l)::≼(G, lT); G,h⊢x::≼T'; np a' x' = None|] ==>
  h''≤|h(a↦(c,(fs((fn,fd)↦x)))) ∧
  Norm(h(a↦(c,(fs((fn,fd)↦x)))), l)::≼(G, lT) ∧
  G,h(a↦(c,(fs((fn,fd)↦x))))⊢x::≼T'"
apply( drule np_NoneD)
apply( erule conjE)
apply( simp)
apply( drule non_np_objD)
apply(  assumption)
apply( clarify)
apply( simp (no_asm_use))
apply( frule (1) hext_objD)
apply( erule exE)
apply( simp)
apply( clarify)
apply( rule conjI)
apply(  fast elim: hext_trans hext_upd_obj)
apply( rule conjI)
prefer 2
apply(  fast elim: conf_upd_obj [THEN iffD2])

apply( rule conforms_upd_obj)
apply   auto
apply(  rule_tac [2] hextI)
prefer 2
apply(  force)
apply( rule oconf_hext)
apply(  erule_tac [2] hext_upd_obj)
apply (frule wf_prog_ws_prog)
apply( drule (2) widen_cfs_fields)
apply( rule oconf_obj [THEN iffD2])
apply( simp (no_asm))
apply( intro strip)
apply( case_tac "(aaa, b) = (fn, fd)")
apply(  simp)
apply(  fast intro: conf_widen)
apply( fast dest: conforms_heapD [THEN hconfD] oconf_objD)
done



lemma Call_lemma2: "[| wf_prog wf_mb G; list_all2 (conf G h) pvs pTs;
   list_all2 (λT T'. G⊢T≼T') pTs pTs'; wf_mhead G (mn,pTs') rT;
  length pTs' = length pns; distinct pns;
  Ball (set lvars) (split (λvn. is_type G))
  |] ==> G,h⊢init_vars lvars(pns[↦]pvs)[::≼]map_of lvars(pns[↦]pTs')"
apply (unfold wf_mhead_def)
apply( clarsimp)
apply( rule lconf_ext_list)
apply(    rule Ball_set_table [THEN lconf_init_vars])
```

```
apply(   force)
apply(  assumption)
apply( assumption)
apply( erule (2) conf_list_gext_widen)
done

lemma Call_type_sound:
 "[| wf_java_prog G; a' ≠ Null; Norm (h, l)::⪯(G, lT); class G C = Some y;
     max_spec G C (mn,pTsa) = {((mda,rTa),pTs')}; xc≤|xh; xh≤|h;
     list_all2 (conf G h) pvs pTsa;
     (md, rT, pns, lvars, blk, res) =
               the (method (G,fst (the (h (the_Addr a')))) (mn, pTs'));
  ∀lT. (np a' None, h, init_vars lvars(pns[↦]pvs)(This↦a'))::⪯(G, lT) -->
  (G, lT)⊢blk√ -->  h≤|xi ∧  (xcptb, xi, xl)::⪯(G, lT);
  ∀lT. (xcptb,xi, xl)::⪯(G, lT) --> (∀T. (G, lT)⊢res::T -->
          xi≤|h' ∧ (x',h', xj)::⪯(G, lT) ∧ (x' = None --> G,h'⊢v::⪯T));
  G,xh⊢a'::⪯ Class C
  |] ==>
  xc≤|h' ∧ (x',(h', l))::⪯(G, lT) ∧  (x' = None --> G,h'⊢v::⪯rTa)"
apply( drule max_spec2mheads)
apply( clarify)
apply( drule (2) non_np_objD')
apply( clarsimp)
apply( frule (1) hext_objD)
apply( clarsimp)
apply( drule (3) Call_lemma)
apply( clarsimp simp add: wf_java_mdecl_def)
apply( erule_tac V = "method ?sig ?x = ?y" in thin_rl)
apply( drule spec, erule impE,  erule_tac [2] notE impE, tactic "assume_tac 2")
apply(  rule conformsI)
apply(   erule conforms_heapD)
apply(  rule lconf_ext)
apply(   force elim!: Call_lemma2)
apply(  erule conf_hext, erule (1) conf_obj_AddrI)
apply( erule_tac V = "?E⊢?blk√" in thin_rl)
apply (simp add: conforms_def)

apply( erule conjE)
apply( drule spec, erule (1) impE)
apply( drule spec, erule (1) impE)
apply( erule_tac V = "?E⊢res::?rT" in thin_rl)
apply( clarify)
apply( rule conjI)
apply(  fast intro: hext_trans)
apply( rule conjI)
apply(  rule_tac [2] impI)
apply(  erule_tac [2] notE impE, tactic "assume_tac 2")
apply(  frule_tac [2] conf_widen)
apply(    tactic "assume_tac 4")
apply(   tactic "assume_tac 2")
prefer 2
apply(  fast elim!: widen_trans)
apply (rule conforms_xcpt_change)
apply( rule conforms_hext) apply assumption
```

```
apply( erule (1) hext_trans)
apply( erule conforms_heapD)
apply (simp add: conforms_def)
done
```

```
declare split_if [split del]
declare fun_upd_apply [simp del]
declare fun_upd_same [simp]
declare wf_prog_ws_prog [simp]
```

```
ML{*
val forward_hyp_tac = ALLGOALS (TRY o (EVERY' [dtac spec, mp_tac,
  (mp_tac ORELSE' (dtac spec THEN' mp_tac)), REPEAT o (etac conjE)]))
*}
```

```
theorem eval_evals_exec_type_sound:
"wf_java_prog G ==>
  (G⊢(x,(h,l)) -e ≻v -> (x', (h',l')) -->
      (∀lT.   (x,(h ,l ))::≼(G,lT) --> (∀T . (G,lT)⊢e :: T -->
      h≤|h' ∧ (x',(h',l'))::≼(G,lT) ∧ (x'=None --> G,h'⊢v ::≼ T )))) ∧
  (G⊢(x,(h,l)) -es[≻]vs-> (x', (h',l')) -->
      (∀lT.   (x,(h ,l ))::≼(G,lT) --> (∀Ts. (G,lT)⊢es[::]Ts -->
      h≤|h' ∧ (x',(h',l'))::≼(G,lT) ∧ (x'=None --> list_all2 (λv T. G,h'⊢v::≼T) vs
Ts)))) ∧
  (G⊢(x,(h,l)) -c       -> (x', (h',l')) -->
      (∀lT.   (x,(h ,l ))::≼(G,lT) -->      (G,lT)⊢c  √ -->
      h≤|h' ∧ (x',(h',l'))::≼(G,lT)))"
apply( rule eval_evals_exec_induct)
apply( unfold c_hupd_def)
```

— several simplifications, XcptE, XcptEs, XcptS, Skip, Nil??

```
apply( simp_all)
apply( tactic "ALLGOALS strip_tac")
apply( tactic {* ALLGOALS (eresolve_tac [thm "ty_expr.cases", thm "ty_exprs.cases", thm
"wt_stmt.cases"]
                 THEN_ALL_NEW (full_simp_tac (global_simpset_of @{theory Conform}))) *})
apply(tactic "ALLGOALS (EVERY' [REPEAT o (etac conjE), REPEAT o hyp_subst_tac])")
```

— Level 7

— 15 NewC

```
apply (drule sym)
apply( drule new_AddrD)
apply( erule disjE)
prefer 2
apply( simp (no_asm_simp))
apply (rule conforms_xcpt_change, assumption)
apply (simp (no_asm_simp) add: xconf_def)
apply( clarsimp)
apply( rule conjI)
apply( force elim!: NewC_conforms)
apply( rule conf_obj_AddrI)
```

```
apply(  rule_tac [2] rtrancl.rtrancl_refl)
apply( simp (no_asm))
```

— for Cast

```
defer 1
```

— 14 Lit

```
apply( erule conf_litval)
```

— 13 BinOp

```
apply (tactic "forward_hyp_tac")
apply (tactic "forward_hyp_tac")
apply( rule conjI, erule (1) hext_trans)
apply( erule conjI)
apply( clarsimp)
apply( drule eval_no_xcpt)
apply( simp split add: binop.split)
```

— 12 LAcc

```
apply simp
apply( fast elim: conforms_localD [THEN lconfD])
```

— for FAss

```
apply( tactic {* EVERY'[eresolve_tac [thm "ty_expr.cases", thm "ty_exprs.cases", thm "wt_stmt.cases"]

       THEN_ALL_NEW (full_simp_tac @{simpset}), REPEAT o (etac conjE), hyp_subst_tac]
3*})
```

— for if

```
apply( tactic {* (InductTacs.case_tac @{context} "the_Bool v" THEN_ALL_NEW
  (asm_full_simp_tac @{simpset})) 7*})

apply (tactic "forward_hyp_tac")
```

— 11+1 if

```
prefer 7
apply(  fast intro: hext_trans)
prefer 7
apply(  fast intro: hext_trans)
```

— 10 Expr

```
prefer 6
apply( fast)
```

— 9 ???

```
apply( simp_all)
```

— 8 Cast

```
prefer 8
apply (rule conjI)
  apply (fast intro: conforms_xcpt_change xconf_raise_if)

  apply clarify
  apply (drule raise_if_NoneD)
```

```
  apply (clarsimp)
  apply (rule Cast_conf)
  apply assumption+


— 7 LAss
apply (fold fun_upd_def)
apply( tactic {* (eresolve_tac [thm "ty_expr.cases", thm "ty_exprs.cases", thm "wt_stmt.cases"]

                 THEN_ALL_NEW (full_simp_tac @{simpset})) 1 *})
apply (intro strip)
apply (case_tac E)
apply (simp)
apply( blast intro: conforms_upd_local conf_widen)

— 6 FAcc
apply (rule conjI)
  apply (simp add: np_def)
  apply (fast intro: conforms_xcpt_change xconf_raise_if)
apply( fast elim!: FAcc_type_sound)

— 5 While
prefer 5
apply(erule_tac V = "?a ⟶ ?b" in thin_rl)
apply(drule (1) ty_expr_ty_exprs_wt_stmt.Loop)
apply(force elim: hext_trans)

apply (tactic "forward_hyp_tac")

— 4 Cond
prefer 4
apply (case_tac "the_Bool v")
apply simp
apply( fast dest: evals_no_xcpt intro: conf_hext hext_trans)
apply simp
apply( fast dest: evals_no_xcpt intro: conf_hext hext_trans)

— 3 ;;
prefer 3
apply( fast dest: evals_no_xcpt intro: conf_hext hext_trans)


— 2 FAss
apply (subgoal_tac "(np a' x1, aa, ba) ::≼ (G, lT)")
  prefer 2
  apply (simp add: np_def)
  apply (fast intro: conforms_xcpt_change xconf_raise_if)
apply( case_tac "x2")
  — x2 = None
  apply (simp)
  apply (tactic forward_hyp_tac, clarify)
  apply( drule eval_no_xcpt)
  apply( erule FAss_type_sound, rule HOL.refl, assumption+)
  — x2 = Some a
```

```
  apply (  simp (no_asm_simp))
  apply(  fast intro: hext_trans)


apply( tactic prune_params_tac)
— Level 52

— 1 Call
apply( case_tac "x")
prefer 2
apply(  clarsimp)
apply(  drule exec_xcpt)
apply(  simp)
apply(  drule_tac eval_xcpt)
apply(  simp)
apply(  fast elim: hext_trans)
apply( clarify)
apply( drule evals_no_xcpt)
apply( simp)
apply( case_tac "a' = Null")
apply(  simp)
apply(  drule exec_xcpt)
apply(  simp)
apply(  drule eval_xcpt)
apply(  simp)
apply (rule conjI)
  apply(  fast elim: hext_trans)
  apply (rule conforms_xcpt_change, assumption)
  apply (simp (no_asm_simp) add: xconf_def)
apply(clarsimp)

apply( drule ty_expr_is_type, simp)
apply(clarsimp)
apply(unfold is_class_def)
apply(clarsimp)

apply(rule Call_type_sound)
prefer 11
apply blast
apply (simp (no_asm_simp))+

done


lemma eval_type_sound: "!!E s s'.
  [| wf_java_prog G; G⊢(x,s) -e≻v -> (x',s'); (x,s)::≼E; E⊢e::T; G=prg E |]
  ==> (x',s')::≼E ∧ (x'=None --> G,heap s'⊢v::≼T) ∧ heap s ≤| heap s'"
apply (simp (no_asm_simp) only: split_tupled_all)
apply (drule eval_evals_exec_type_sound [THEN conjunct1, THEN mp, THEN spec, THEN mp])
apply auto
done


lemma evals_type_sound: "!!E s s'.
```

```
  [| wf_java_prog G; G⊢(x,s) -es[≻]vs -> (x',s'); (x,s)::≼E; E⊢es[::]Ts; G=prg E |]

  ==> (x',s')::≼E ∧ (x'=None --> (list_all2 (λv T. G,heap s'⊢v::≼T) vs Ts)) ∧ heap
s ≤| heap s'"
apply (simp (no_asm_simp) only: split_tupled_all)
apply (drule eval_evals_exec_type_sound [THEN conjunct2, THEN conjunct1, THEN mp, THEN
spec, THEN mp])
apply auto
done

lemma exec_type_sound: "!!E s s'.
  ⟦ wf_java_prog G; G⊢(x,s) -s0-> (x',s'); (x,s)::≼E; E⊢s0√; G=prg E ⟧
  ⟹ (x',s')::≼E ∧ heap s ≤| heap s'"
apply( simp (no_asm_simp) only: split_tupled_all)
apply (drule eval_evals_exec_type_sound
             [THEN conjunct2, THEN conjunct2, THEN mp, THEN spec, THEN mp])
apply   auto
done

theorem all_methods_understood:
"[|G=prg E; wf_java_prog G; G⊢(x,s) -e≻a'-> Norm s'; a' ≠ Null;
          (x,s)::≼E; E⊢e::Class C; method (G,C) sig ≠ None|] ==>
  method (G,fst (the (heap s' (the_Addr a')))) sig ≠ None"
apply (frule eval_type_sound, assumption+)
apply(clarsimp)
apply( frule widen_methd)
apply assumption
prefer 2
apply(  fast)
apply( drule non_npD)
apply auto
done

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

end
```

2.14 Example MicroJava Program

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

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

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

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

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

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

consts
  cnam' :: "cnam' => cname"
  vnam' :: "vnam' => vnam"

— *cnam'* and *vnam'* are intended to be isomorphic to *cnam* and *vnam*
axioms
  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"

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

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

```
  where "e == VName (vnam' e')"

axioms
  Base_not_Object: "Base ≠ Object"
  Ext_not_Object:  "Ext  ≠ Object"
  Base_not_Xcpt:   "Base ≠ Xcpt z"
  Ext_not_Xcpt:    "Ext  ≠ Xcpt z"
  e_not_This:      "e ≠ This"

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

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

defs
  foo_Base_def:"foo_Base == ([x],[],Skip,LAcc x)"
  BaseC_def:"BaseC == (Base, (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 Numeral1)),
          Lit Null)"
  ExtC_def: "ExtC  == (Ext,  (Base  ,
          [(vee, PrimT Integer)],
          [((foo,[Class Base]),Class Ext,foo_Ext)]))"

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


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

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

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

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

lemmas *map_of_Cons = map_of.simps(2)*

lemma *map_of_Cons1 [simp]: "map_of ((aa,bb)#ps) aa = Some bb"*
apply *(simp (no_asm))*
done
lemma *map_of_Cons2 [simp]: "aa≠k ==> map_of ((k,bb)#ps) aa = map_of ps aa"*
apply *(simp (no_asm_simp))*
done
declare *map_of_Cons [simp del]* — sic!

lemma *class_tprg_Object [simp]: "class tprg Object = Some (undefined, [], [])"*
apply *(unfold ObjectC_def class_def)*
apply *(simp (no_asm))*
done

lemma *class_tprg_NP [simp]: "class tprg (Xcpt NP) = Some (Object, [], [])"*
apply *(unfold ObjectC_def NullPointerC_def ClassCastC_def OutOfMemoryC_def BaseC_def ExtC_def class_def)*
apply *(simp (no_asm))*
done

lemma *class_tprg_OM [simp]: "class tprg (Xcpt OutOfMemory) = Some (Object, [], [])"*
apply *(unfold ObjectC_def NullPointerC_def ClassCastC_def OutOfMemoryC_def BaseC_def ExtC_def class_def)*
apply *(simp (no_asm))*
done

lemma *class_tprg_CC [simp]: "class tprg (Xcpt ClassCast) = Some (Object, [], [])"*
apply *(unfold ObjectC_def NullPointerC_def ClassCastC_def OutOfMemoryC_def BaseC_def ExtC_def class_def)*
apply *(simp (no_asm))*
done

lemma *class_tprg_Base [simp]:*
"class tprg Base = Some (Object,
    [(vee, PrimT Boolean)],
        [((foo, [Class Base]), Class Base, foo_Base)])"
apply *(unfold ObjectC_def NullPointerC_def ClassCastC_def OutOfMemoryC_def BaseC_def ExtC_def class_def)*
apply *(simp (no_asm))*
done

lemma *class_tprg_Ext [simp]:*
"class tprg Ext = Some (Base,
    [(vee, PrimT Integer)],
        [((foo, [Class Base]), Class Ext, foo_Ext)])"
apply *(unfold ObjectC_def BaseC_def ExtC_def class_def)*
apply *(simp (no_asm))*

done

lemma *not_Object_subcls [elim!]: "(Object, C) ∈ (subcls1 tprg)^+ ==> R"*
apply *(auto dest!: tranclD subcls1D)*
done

lemma *subcls_ObjectD [dest!]: "tprg⊢Object≼C C ==> C = Object"*
apply *(erule rtrancl_induct)*
apply *auto*
apply *(drule subcls1D)*
apply *auto*
done

lemma *not_Base_subcls_Ext [elim!]: "(Base, Ext) ∈ (subcls1 tprg)^+ ==> R"*
apply *(auto dest!: tranclD subcls1D)*
done

lemma *class_tprgD:*
"class tprg C = Some z ==> C=Object ∨ C=Base ∨ C=Ext ∨ C=Xcpt NP ∨ C=Xcpt ClassCast ∨ C=Xcpt OutOfMemory"
apply *(unfold ObjectC_def ClassCastC_def NullPointerC_def OutOfMemoryC_def BaseC_def ExtC_def class_def)*
apply *(auto split add: split_if_asm simp add: map_of_Cons)*
done

lemma *not_class_subcls_class [elim!]: "(C, C) ∈ (subcls1 tprg)^+ ==> R"*
apply *(auto dest!: tranclD subcls1D)*
apply *(frule class_tprgD)*
apply *(auto dest!:)*
apply *(drule rtranclD)*
apply *auto*
done

lemma *unique_classes: "unique tprg"*
apply *(simp (no_asm) add: ObjectC_def BaseC_def ExtC_def NullPointerC_def ClassCastC_def OutOfMemoryC_def)*
done

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

lemma *Ext_subcls_Base [simp]: "tprg⊢Ext≼C Base"*
apply *(rule subcls_direct)*
apply *auto*
done

lemma *Ext_widen_Base [simp]: "tprg⊢Class Ext≼ Class Base"*
apply *(rule widen.subcls)*
apply *(simp (no_asm))*
done

declare *ty_expr_ty_exprs_wt_stmt.intros [intro!]*

lemma *acyclic_subcls1': "acyclic (subcls1 tprg)"*
apply *(rule acyclicI)*

```
apply safe
done

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

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

lemma fields_Object [simp]: "fields (tprg, Object) = []"
apply (subst fields_rec')
apply  auto
done

declare is_class_def [simp]

lemma fields_Base [simp]: "fields (tprg,Base) = [((vee, Base), PrimT Boolean)]"
apply (subst fields_rec')
apply  auto
done

lemma fields_Ext [simp]:
  "fields (tprg, Ext)  = [((vee, Ext ), PrimT Integer)] @ fields (tprg, Base)"
apply (rule trans)
apply  (rule fields_rec')
apply  auto
done

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

lemma method_Object [simp]: "method (tprg,Object) = map_of []"
apply (subst method_rec')
apply  auto
done

lemma method_Base [simp]: "method (tprg, Base) = map_of
  [((foo, [Class Base]), Base, (Class Base, foo_Base))]"
apply (rule trans)
apply  (rule method_rec')
apply  auto
done

lemma method_Ext [simp]: "method (tprg, Ext) = (method (tprg, Base) ++ map_of
  [((foo, [Class Base]), Ext , (Class Ext, foo_Ext))])"
apply (rule trans)
apply  (rule method_rec')
apply  auto
done

lemma wf_foo_Base:
"wf_mdecl wf_java_mdecl tprg Base ((foo, [Class Base]), (Class Base, foo_Base))"
apply (unfold wf_mdecl_def wf_mhead_def wf_java_mdecl_def foo_Base_def)
apply auto
done

lemma wf_foo_Ext:
```

```
"wf_mdecl wf_java_mdecl tprg Ext ((foo, [Class Base]), (Class Ext, foo_Ext))"
apply (unfold wf_mdecl_def wf_mhead_def wf_java_mdecl_def foo_Ext_def)
apply auto
apply  (rule ty_expr_ty_exprs_wt_stmt.Cast)
prefer 2
apply   (simp)
apply   (rule_tac [2] cast.subcls)
apply   (unfold field_def)
apply   auto
done

lemma wf_ObjectC:
"ws_cdecl tprg ObjectC ∧
  wf_cdecl_mdecl wf_java_mdecl tprg ObjectC ∧ wf_mrT tprg ObjectC"
apply (unfold ws_cdecl_def wf_cdecl_mdecl_def
  wf_mrT_def wf_fdecl_def ObjectC_def)
apply (simp (no_asm))
done

lemma wf_NP:
"ws_cdecl tprg NullPointerC ∧
  wf_cdecl_mdecl wf_java_mdecl tprg NullPointerC ∧ wf_mrT tprg NullPointerC"
apply (unfold ws_cdecl_def wf_cdecl_mdecl_def
  wf_mrT_def wf_fdecl_def NullPointerC_def)
apply (simp add: class_def)
apply (fold NullPointerC_def class_def)
apply auto
done

lemma wf_OM:
"ws_cdecl tprg OutOfMemoryC ∧
  wf_cdecl_mdecl wf_java_mdecl tprg OutOfMemoryC ∧ wf_mrT tprg OutOfMemoryC"
apply (unfold ws_cdecl_def wf_cdecl_mdecl_def
  wf_mrT_def wf_fdecl_def OutOfMemoryC_def)
apply (simp add: class_def)
apply (fold OutOfMemoryC_def class_def)
apply auto
done

lemma wf_CC:
"ws_cdecl tprg ClassCastC ∧
  wf_cdecl_mdecl wf_java_mdecl tprg ClassCastC ∧ wf_mrT tprg ClassCastC"
apply (unfold ws_cdecl_def wf_cdecl_mdecl_def
  wf_mrT_def wf_fdecl_def ClassCastC_def)
apply (simp add: class_def)
apply (fold ClassCastC_def class_def)
apply auto
done

lemma wf_BaseC:
"ws_cdecl tprg BaseC ∧
  wf_cdecl_mdecl wf_java_mdecl tprg BaseC ∧ wf_mrT tprg BaseC"
apply (unfold ws_cdecl_def wf_cdecl_mdecl_def
  wf_mrT_def wf_fdecl_def BaseC_def)
```

```
apply (simp (no_asm))
apply (fold BaseC_def)
apply (rule mp) defer apply (rule wf_foo_Base)
apply (auto simp add: wf_mdecl_def)
done


lemma wf_ExtC:
"ws_cdecl tprg ExtC ∧
  wf_cdecl_mdecl wf_java_mdecl tprg ExtC ∧ wf_mrT tprg ExtC"
apply (unfold ws_cdecl_def wf_cdecl_mdecl_def
  wf_mrT_def wf_fdecl_def ExtC_def)
apply (simp (no_asm))
apply (fold ExtC_def)
apply (rule mp) defer apply (rule wf_foo_Ext)
apply (auto simp add: wf_mdecl_def)
apply (drule rtranclD)
apply auto
done


lemma [simp]: "fst ObjectC = Object" by (simp add: ObjectC_def)

lemma wf_tprg:
"wf_prog wf_java_mdecl tprg"
apply (unfold wf_prog_def ws_prog_def Let_def)
apply (simp add: wf_ObjectC wf_BaseC wf_ExtC wf_NP wf_OM wf_CC unique_classes)
apply (rule wf_syscls)
apply (simp add: SystemClasses_def)
done

lemma appl_methds_foo_Base:
"appl_methds tprg Base (foo, [NT]) =
  {((Class Base, Class Base), [Class Base])}"
apply (unfold appl_methds_def)
apply (simp (no_asm))
done

lemma max_spec_foo_Base: "max_spec tprg Base (foo, [NT]) =
  {((Class Base, Class Base), [Class Base])}"
apply (unfold max_spec_def)
apply (auto simp add: appl_methds_foo_Base)
done

ML {* val t = resolve_tac @{thms ty_expr_ty_exprs_wt_stmt.intros} 1 *}
schematic_lemma wt_test: "(tprg, empty(e↦Class Base))⊢
  Expr(e::=NewC Ext);; Expr({Base}LAcc e..foo({?pTs'}[Lit Null]))√"
apply (tactic t) — ;;
apply  (tactic t) — Expr
apply  (tactic t) — LAss
apply    simp — e ≠ This
apply    (tactic t) — LAcc
apply     (simp (no_asm))
apply    (simp (no_asm))
```

```
apply   (tactic t) — NewC
apply   (simp (no_asm))
apply  (simp (no_asm))
apply (tactic t) — Expr
apply (tactic t) — Call
apply   (tactic t) — LAcc
apply    (simp (no_asm))
apply   (simp (no_asm))
apply  (tactic t) — Cons
apply   (tactic t) — Lit
apply   (simp (no_asm))
apply  (tactic t) — Nil
apply (simp (no_asm))
apply (rule max_spec_foo_Base)
done

ML {* val e = resolve_tac (@{thm NewCI} :: @{thms eval_evals_exec.intros}) 1 *}

declare split_if [split del]
declare init_vars_def [simp] c_hupd_def [simp] cast_ok_def [simp]
schematic_lemma exec_test:
" [|new_Addr (heap (snd s0)) = (a, None)|] ==>
  tprg⊢s0 -test-> ?s"
apply (unfold test_def)
— ?s = s3
apply (tactic e) — ;;
apply  (tactic e) — Expr
apply  (tactic e) — LAss
apply   (tactic e) — NewC
apply    force
apply   force
apply  (simp (no_asm))
apply (erule thin_rl)
apply (tactic e) — Expr
apply (tactic e) — Call
apply       (tactic e) — LAcc
apply      force
apply     (tactic e) — Cons
apply      (tactic e) — Lit
apply     (tactic e) — Nil
apply    (simp (no_asm))
apply   (force simp add: foo_Ext_def)
apply  (simp (no_asm))
apply  (tactic e) — Expr
apply  (tactic e) — FAss
apply       (tactic e) — Cast
apply        (tactic e) — LAcc
apply       (simp (no_asm))
apply      (simp (no_asm))
apply     (simp (no_asm))
apply     (tactic e) — XcptE
apply    (simp (no_asm))
apply   (rule surjective_pairing [THEN sym, THEN[2]trans], subst Pair_eq, force)
apply  (simp (no_asm))
```

apply *(simp (no_asm))*
apply *(tactic e)* — XcptE
done

end

2.15 Example for generating executable code from Java semantics

```
theory JListExample
imports Eval
begin

ML {* Syntax.ambiguity_level := 100000 *}

consts
  list_name :: cname
  append_name :: mname
  val_nam :: vnam
  next_nam :: vnam
  l_nam :: vnam
  l1_nam :: vnam
  l2_nam :: vnam
  l3_nam :: vnam
  l4_nam :: vnam

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

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

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

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

definition l2_name :: vname where
  "l2_name == VName l2_nam"

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

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

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

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

types_code
  cname ("string")
  vnam ("string")
  mname ("string")
  loc' ("int")

consts_code
  "new_Addr" ("⟨module⟩new'_addr {* %x. case x of None => True | Some y => False *}/ {*
None *} {* Loc *}")
attach {*
fun new_addr p none loc hp =
  let fun nr i = if p (hp (loc i)) then (loc i, none) else nr (i+1);
  in nr 0 end;
*}

  "undefined" ("(raise Match)")
  "undefined :: val" ("{* Unit *}")
  "undefined :: cname" ("""")

  "Object" (""Object"")
  "list_name" (""list"")
  "append_name" (""append"")
  "val_nam" (""val"")
  "next_nam" (""next"")
  "l_nam" (""l"")
  "l1_nam" (""l1"")
  "l2_nam" (""l2"")
  "l3_nam" (""l3"")
  "l4_nam" (""l4"")

code_module J
contains
  test = "example_prg⊢Norm (empty, empty)
    -(Expr (l1_name::=NewC list_name);;
      Expr ({list_name}(LAcc l1_name)..val_name:=Lit (Intg 1));;
      Expr (l2_name::=NewC list_name);;
      Expr ({list_name}(LAcc l2_name)..val_name:=Lit (Intg 2));;
      Expr (l3_name::=NewC list_name);;
      Expr ({list_name}(LAcc l3_name)..val_name:=Lit (Intg 3));;
      Expr (l4_name::=NewC list_name);;
      Expr ({list_name}(LAcc l4_name)..val_name:=Lit (Intg 4));;
      Expr ({list_name}(LAcc l1_name)..
        append_name({[RefT (ClassT list_name)]}[LAcc l2_name]));;
      Expr ({list_name}(LAcc l1_name)..
        append_name({[RefT (ClassT list_name)]}[LAcc l3_name]));;
      Expr ({list_name}(LAcc l1_name)..
        append_name({[RefT (ClassT list_name)]}[LAcc l4_name])))-> _"
```

2.15.1 Big step execution

```
ML {*
```

```
val SOME ((_, (heap, locs)), _) = DSeq.pull J.test;
locs J.l1_name;
locs J.l2_name;
locs J.l3_name;
locs J.l4_name;
snd (J.the (heap (J.Loc 0))) (J.val_name, "list");
snd (J.the (heap (J.Loc 0))) (J.next_name, "list");
snd (J.the (heap (J.Loc 1))) (J.val_name, "list");
snd (J.the (heap (J.Loc 1))) (J.next_name, "list");
snd (J.the (heap (J.Loc 2))) (J.val_name, "list");
snd (J.the (heap (J.Loc 2))) (J.next_name, "list");
snd (J.the (heap (J.Loc 3))) (J.val_name, "list");
snd (J.the (heap (J.Loc 3))) (J.next_name, "list");

*}
```

end

Chapter 3

Java Virtual Machine

3.1 State of the JVM

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

3.1.1 Frame Stack

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

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

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

3.1.2 Exceptions

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

3.1.3 Runtime State

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

3.1.4 Lemmas

```
lemma new_Addr_OutOfMemory:
  "snd (new_Addr hp) = Some xcp ⟹ xcp = Addr (XcptRef OutOfMemory)"
proof -
  obtain ref xp where "new_Addr hp = (ref, xp)" by (cases "new_Addr hp")
  moreover
  assume "snd (new_Addr hp) = Some xcp"
  ultimately
  show ?thesis by (auto dest: new_AddrD)
qed

end
```

3.2 Instructions of the JVM

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

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

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

end
```

3.3 JVM Instruction Semantics

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

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

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

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

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

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

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

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

 "exec_instr (Invoke C mn ps) G hp stk vars Cl sig pc frs =
  (let n    = length ps;
```

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

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

```
"exec_instr Return G hp stk0 vars Cl sig0 pc frs =
  (if frs=[] then
     (None, hp, [])
   else
     let val = hd stk0; (stk,loc,C,sig,pc) = hd frs;
         (mn,pt) = sig0; n = length pt
   in
      (None, hp, (val#(drop (n+1) stk),loc,C,sig,pc+1)#tl frs))"
```

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

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

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

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

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

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

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

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

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

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

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

end

3.4 Exception handling in the JVM

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

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


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

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


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

System exceptions are allocated in all heaps:

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

```
lemma match_exception_table_in_et:
  "match_exception_table G C pc et = Some pc' ⟹ ∃e ∈ set et. pc' = fst (snd (snd e))"
  by (induct et) (auto split: split_if_asm)


end
```

3.5 Program Execution in the JVM

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

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

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

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

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

notation *(xsymbols)*
```
  exec_all  ("_ ⊢ _ -jvm→ _" [61,61,61]60)
```

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

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

end

3.6 Example for generating executable code from JVM semantics

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

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

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

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

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

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

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

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

definition make_list_ins :: bytecode where
  "make_list_ins ==
       [New list_name,
```

```
        Dup,
        Store 0,
        LitPush (Intg 1),
        Putfield val_name list_name,
        New list_name,
        Dup,
        Store 1,
        LitPush (Intg 2),
        Putfield val_name list_name,
        New list_name,
        Dup,
        Store 2,
        LitPush (Intg 3),
        Putfield val_name list_name,
        Load 0,
        Load 1,
        Invoke list_name append_name [Class list_name],
        Pop,
        Load 0,
        Load 2,
        Invoke list_name append_name [Class list_name],
        Return]"
```

definition *test_class :: "jvm_method class"* **where**

```
  "test_class ==
    (Object, [],
     [((makelist_name, []), PrimT Void, (3, 2, make_list_ins,[]))])"
```

definition *E :: jvm_prog* **where**

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

types_code

```
  cnam ("string")
  vnam ("string")
  mname ("string")
  loc' ("int")
```

consts_code

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

attach *{**

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

  "undefined" ("(raise Match)")
  "undefined :: val" ("{* Unit *}")
  "undefined :: cname" ("{* Object *}")

  "list_nam" (""list"")
  "test_nam" (""test"")
  "append_name" (""append"")
```

```
  "makelist_name" (""makelist"")
  "val_nam" (""val"")
  "next_nam" (""next"")
```

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

3.6.1 Single step execution

code_module *JVM*
contains
```
  exec = exec
  test = test
```

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

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

end

3.7 A Defensive JVM

```
theory JVMDefensive
imports JVMExec
begin
```

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

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


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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

definition *check :: "jvm_prog ⇒ jvm_state ⇒ bool"* **where**

```
  "check G s ≡ let (xcpt, hp, frs) = s in
             (case frs of [] ⇒ True | (stk,loc,C,sig,pc)#frs' ⇒
              (let  (C',rt,mxs,mxl,ins,et) = the (method (G,C) sig); i = ins!pc in
               pc < size ins ∧
               check_instr i G hp stk loc C sig pc mxs frs'))"
```

definition *exec_d :: "jvm_prog ⇒ jvm_state type_error ⇒ jvm_state option type_error"*

```
where
  "exec_d G s ≡ case s of
      TypeError ⇒ TypeError
    | Normal s' ⇒ if check G s' then Normal (exec (G, s')) else TypeError"


constdefs
  exec_all_d :: "jvm_prog ⇒ jvm_state type_error ⇒ jvm_state type_error ⇒ bool"
                 ("_ |- _ -jvmd-> _" [61,61,61]60)
  "G |- s -jvmd-> t ≡
        (s,t) ∈ ({(s,t). exec_d G s = TypeError ∧ t = TypeError} ∪
                {(s,t). ∃t'. exec_d G s = Normal (Some t') ∧ t = Normal t'})*"

notation (xsymbols)
  exec_all_d  ("_ ⊢ _ -jvmd→ _" [61,61,61]60)


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

lemma [dest!]:
  "(if P then A else B) ≠ B ⟹ P"
  by (cases P, auto)

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

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


lemma defensive_imp_aggressive:
  "G ⊢ (Normal s) -jvmd→ (Normal t) ⟹ G ⊢ s -jvm→ t"
proof -
  have "⋀x y. G ⊢ x -jvmd→ y ⟹ ∀s t. x = Normal s ⟶ y = Normal t ⟶  G ⊢ s -jvm→
t"
    apply (unfold exec_all_d_def)
    apply (erule rtrancl_induct)
     apply (simp add: exec_all_def)
    apply (fold exec_all_d_def)
    apply simp
    apply (intro allI impI)
    apply (erule disjE, simp)
    apply (elim exE conjE)
    apply (erule allE, erule impE, assumption)
    apply (simp add: exec_all_def exec_d_def split: type_error.splits split_if_asm)
    apply (rule rtrancl_trans, assumption)
    apply blast
    done
  moreover
  assume "G ⊢ (Normal s) -jvmd→ (Normal t)"
```

```
  ultimately
  show "G ⊢ s -jvm→ t" by blast
qed

end
```

Chapter 4

Bytecode Verifier

4.1 Semilattices

```
theory Semilat
imports Main While_Combinator
begin

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

consts
  "lesub" :: "'a ⇒ 'a ord ⇒ 'a ⇒ bool"
  "lesssub" :: "'a ⇒ 'a ord ⇒ 'a ⇒ bool"
  "plussub" :: "'a ⇒ ('a ⇒ 'b ⇒ 'c) ⇒ 'b ⇒ 'c"notation (xsymbols)
  "lesub"  ("(_ /⊑_ _)" [50, 0, 51] 50) and
  "lesssub"  ("(_ /⊏_ _)" [50, 0, 51] 50) and
  "plussub"  ("(_ /⊔_ _)" [65, 0, 66] 65)
defs
  lesub_def:   "x ⊑r y ≡ r x y"
  lesssub_def: "x ⊏r y ≡ x ⊑r y ∧ x ≠ y"
  plussub_def: "x ⊔f y ≡ f x y"

definition ord :: "('a × 'a) set ⇒ 'a ord" where
  "ord r ≡ λx y. (x,y) ∈ r"

definition order :: "'a ord ⇒ bool" where
  "order r ≡ (∀x. x ⊑r x) ∧ (∀x y. x ⊑r y ∧ y ⊑r x ⟶ x=y) ∧ (∀x y z. x ⊑r y ∧
y ⊑r z ⟶ x ⊑r z)"

definition top :: "'a ord ⇒ 'a ⇒ bool" where
  "top r T ≡ ∀x. x ⊑r T"

definition acc :: "'a ord ⇒ bool" where
  "acc r ≡ wf {(y,x). x ⊏r y}"

definition closed :: "'a set ⇒ 'a binop ⇒ bool" where
  "closed A f ≡ ∀x∈A. ∀y∈A. x ⊔f y ∈ A"

definition semilat :: "'a sl ⇒ bool" where
  "semilat ≡ λ(A,r,f). order r ∧ closed A f ∧
                     (∀x∈A. ∀y∈A. x ⊑r x ⊔f y) ∧
                     (∀x∈A. ∀y∈A. y ⊑r x ⊔f y) ∧
                     (∀x∈A. ∀y∈A. ∀z∈A. x ⊑r z ∧ y ⊑r z ⟶ x ⊔f y ⊑r z)"

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

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

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

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

lemma order_refl [simp, intro]: "order r ⟹ x ⊑r x"

lemma order_antisym: "⟦ order r; x ⊑r y; y ⊑r x ⟧ ⟹ x = y"

lemma order_trans: "⟦ order r; x ⊑r y; y ⊑r z ⟧ ⟹ x ⊑r z"

lemma order_less_irrefl [intro, simp]: "order r ⟹ ¬ x ⊏r x"

lemma order_less_trans: "⟦ order r; x ⊏r y; y ⊏r z ⟧ ⟹ x ⊏r z"

lemma topD [simp, intro]: "top r T ⟹ x ⊑r T"

lemma top_le_conv [simp]: "⟦ order r; top r T ⟧ ⟹ (T ⊑r x) = (x = T)"

lemma semilat_Def:
"semilat(A,r,f) ≡ order r ∧ closed A f ∧
                 (∀x∈A. ∀y∈A. x ⊑r x ⊔f y) ∧
                 (∀x∈A. ∀y∈A. y ⊑r x ⊔f y) ∧
                 (∀x∈A. ∀y∈A. ∀z∈A. x ⊑r z ∧ y ⊑r z ⟶ x ⊔f y ⊑r z)"

lemma (in Semilat) orderI [simp, intro]: "order r"

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

lemma closedD: "⟦ closed A f; x∈A; y∈A ⟧ ⟹ x ⊔f y ∈ A"

lemma closed_UNIV [simp]: "closed UNIV f"

lemma (in Semilat) closed_f [simp, intro]: "⟦x ∈ A; y ∈ A⟧  ⟹ x ⊔f y ∈ A"

lemma (in Semilat) refl_r [intro, simp]: "x ⊑r x" by simp

lemma (in Semilat) antisym_r [intro?]: "⟦ x ⊑r y; y ⊑r x ⟧ ⟹ x = y"

lemma (in Semilat) trans_r [trans, intro?]: "⟦x ⊑r y; y ⊑r z⟧ ⟹ x ⊑r z"

lemma (in Semilat) ub1 [simp, intro?]: "⟦ x ∈ A; y ∈ A ⟧ ⟹ x ⊑r x ⊔f y"

lemma (in Semilat) ub2 [simp, intro?]: "⟦ x ∈ A; y ∈ A ⟧ ⟹ y ⊑r x ⊔f y"

lemma (in Semilat) lub [simp, intro?]:
  "⟦ x ⊑r z; y ⊑r z; x ∈ A; y ∈ A; z ∈ A ⟧ ⟹ x ⊔f y ⊑r z"

lemma (in Semilat) plus_le_conv [simp]:
  "⟦ x ∈ A; y ∈ A; z ∈ A ⟧ ⟹ (x ⊔f y ⊑r z) = (x ⊑r z ∧ y ⊑r z)"

lemma (in Semilat) le_iff_plus_unchanged: "⟦ x ∈ A; y ∈ A ⟧ ⟹ (x ⊑r y) = (x ⊔f y
= y)"
```

```
lemma (in Semilat) le_iff_plus_unchanged2: "⟦ x ∈ A; y ∈ A ⟧ ⟹ (x ⊑_r y) = (y ⊔_f x
= y)"

lemma (in Semilat) plus_assoc [simp]:
  assumes a: "a ∈ A" and b: "b ∈ A" and c: "c ∈ A"
  shows "a ⊔_f (b ⊔_f c) = a ⊔_f b ⊔_f c"
lemma (in Semilat) plus_com_lemma:
  "⟦a ∈ A; b ∈ A⟧ ⟹ a ⊔_f b ⊑_r b ⊔_f a"
lemma (in Semilat) plus_commutative:
  "⟦a ∈ A; b ∈ A⟧ ⟹ a ⊔_f b = b ⊔_f a"

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

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

lemma is_ubD:
  "is_ub r x y u ⟹ (x,u) ∈ r ∧ (y,u) ∈ r"


lemma is_lub_bigger1 [iff]:
  "is_lub (r^* ) x y y = ((x,y)∈r^* )"
lemma is_lub_bigger2 [iff]:
  "is_lub (r^* ) x y x = ((y,x)∈r^* )"
lemma extend_lub:
  "⟦ single_valued r; is_lub (r^* ) x y u; (x',x) ∈ r ⟧
  ⟹ EX v. is_lub (r^* ) x' y v"
lemma single_valued_has_lubs [rule_format]:
  "⟦ single_valued r; (x,u) ∈ r^* ⟧ ⟹ (∀y. (y,u) ∈ r^* ⟶
  (EX z. is_lub (r^* ) x y z))"
lemma some_lub_conv:
  "⟦ acyclic r; is_lub (r^* ) x y u ⟧ ⟹ some_lub (r^* ) x y = u"
lemma is_lub_some_lub:
  "⟦ single_valued r; acyclic r; (x,u)∈r^*; (y,u)∈r^* ⟧
  ⟹ is_lub (r^* ) x y (some_lub (r^* ) x y)"
```

4.1.1 An executable lub-finder

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

lemma exec_lub_refl: "exec_lub r f T T = T"
by (simp add: exec_lub_def while_unfold)

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

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

```
"⟦ single_valued r; acyclic r; (x,u):r^*; (y,u):r^*; ∀x y. (x,y) ∈ r ⟶ f x = y ⟧
⟹ is_lub (r^* ) x y (exec_lub r f x y)"
```

end

4.2 The Error Type

```
theory Err
imports Semilat
begin

datatype 'a err = Err | OK 'a

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

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

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

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

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

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

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

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

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

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


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

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

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

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

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

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

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

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

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

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

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

lemma Err_le_conv [iff]:
 "Err <=_(le r) e  = (e = Err)"
  by (simp add: unfold_lesub_err le_def  split: err.split)
```

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

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

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

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

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

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

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

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

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

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

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

4.2.1 lift

lemma *lift_in_errI:*
  "⟦ e : err S; !x:S. e = OK x ⟶ f x : err S ⟧ ⟹ lift f e : err S"
apply *(unfold lift_def)*
apply *(simp split: err.split)*
apply *blast*
done

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

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

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

4.2.2 sup

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

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

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

lemma *Err_sup_eq_OK_conv [iff]:*
  "(Err.sup f ex ey = OK z) = (? x y. ex = OK x & ey = OK y & f x y = z)"
apply *(unfold Err.sup_def lift2_def plussub_def)*
apply *(rule iffI)*
 apply *(simp split: err.split_asm)*
apply *clarify*
apply *simp*
done

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

4.2.3 semilat (err A) (le r) f

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

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

lemma semilat_le_err_OK1:
  "⟦ x:A; y:A; semilat(err A, le r, f); OK x +_f OK y = OK z ⟧
  ⟹ x <=_r z"
apply (rule OK_le_err_OK [THEN iffD1])
apply (erule subst)
apply (simp add: Semilat.ub1 [OF Semilat.intro])
done

lemma semilat_le_err_OK2:
  "⟦ x:A; y:A; semilat(err A, le r, f); OK x +_f OK y = OK z ⟧
  ⟹ y <=_r z"
apply (rule OK_le_err_OK [THEN iffD1])
apply (erule subst)
apply (simp add: Semilat.ub2 [OF Semilat.intro])
done

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

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

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

4.2.4 semilat (err(Union AS))

```
lemma all_bex_swap_lemma [iff]:
  "(!x. (? y:A. x = f y) ⟶ P x) = (!y:A. P(f y))"
  by blast

lemma closed_err_Union_lift2I:
  "⟦ !A:AS. closed (err A) (lift2 f); AS ~= {};
     !A:AS.!B:AS. A~=B ⟶ (!a:A.!b:B. a +_f b = Err) ⟧
  ⟹ closed (err(Union AS)) (lift2 f)"
apply (unfold closed_def err_def)
apply simp
apply clarify
apply simp
apply fast
done
```

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

```
lemma err_semilat_UnionI:
  "⟦ !A:AS. err_semilat(A, r, f); AS ~= {};
     !A:AS.!B:AS. A~=B ⟶ (!a:A.!b:B. ~ a <=_r b & a +_f b = Err) ⟧
  ⟹ err_semilat(Union AS, r, f)"
apply (unfold semilat_def sl_def)
apply (simp add: closed_err_Union_lift2I)
apply (rule conjI)
 apply blast
apply (simp add: err_def)
apply (rule conjI)
 apply clarify
 apply (rename_tac A a u B b)
 apply (case_tac "A = B")
  apply simp
 apply simp
apply (rule conjI)
 apply clarify
 apply (rename_tac A a u B b)
 apply (case_tac "A = B")
  apply simp
 apply simp
apply clarify
apply (rename_tac A ya yb B yd z C c a b)
```

```
apply (case_tac "A = B")
 apply (case_tac "A = C")
  apply simp
 apply (rotate_tac -1)
 apply simp
apply (rotate_tac -1)
apply (case_tac "B = C")
 apply simp
apply (rotate_tac -1)
apply simp
done

end
```

4.3 Fixed Length Lists

```
theory Listn
imports Err
begin

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

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

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

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

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

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

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

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

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

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

lemmas [simp] = set_update_subsetI

lemma unfold_lesub_list:
  "xs <=[r] ys == Listn.le r xs ys"
  by (simp add: lesub_def)

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

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


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

lemma Cons_less_Conss [simp]:
  "order r ⟹
  x#xs <_(Listn.le r) y#ys =
  (x <_r y & xs <=[r] ys  |  x = y & xs <_(Listn.le r) ys)"
apply (unfold lesssub_def)
apply blast
done

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


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

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

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

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

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

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


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

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

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

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

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

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

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

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

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

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

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

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


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

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


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

lemma listn_appendE [elim!]:
```

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


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

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

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

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

lemma nth_plus_list [rule_format, simp]:
```

```
  "!xs ys i. length xs = n ⟶ length ys = n ⟶ i<n ⟶
  (xs +[f] ys)!i = (xs!i) +_f (ys!i)"
apply (induct n)
 apply simp
apply clarify
apply (case_tac xs)
 apply simp
apply (force simp add: nth_Cons split: list.split nat.split)
done


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

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

lemma (in Semilat) plus_list_lub [rule_format]:
shows "!xs ys zs. set xs <= A ⟶ set ys <= A ⟶ set zs <= A
  ⟶ size xs = n & size ys = n ⟶
  xs <=[r] zs & ys <=[r] zs ⟶ xs +[f] ys <=[r] zs"
apply (unfold unfold_lesub_list)
apply (simp add: Listn.le_def list_all2_conv_all_nth)
done

lemma (in Semilat) list_update_incr [rule_format]:
 "x∈ A ⟹ set xs <= A ⟶
  (!i. i<size xs ⟶ xs <=[r] xs[i := x +_f xs!i])"
apply (unfold unfold_lesub_list)
apply (simp add: Listn.le_def list_all2_conv_all_nth)
apply (induct xs)
 apply simp
apply (simp add: in_list_Suc_iff)
apply clarify
apply (simp add: nth_Cons split: nat.split)
done

lemma equals0I_aux:
  "(⋀y. A y ⟹ False) ⟹ A = bot_class.bot"
  by (rule equals0I) (auto simp add: mem_def)

lemma acc_le_listI [intro!]:
  "⟦ order r; acc r ⟧ ⟹ acc(Listn.le r)"
apply (unfold acc_def)
apply (subgoal_tac
 "wf(UN n. {(ys,xs). size xs = n ∧ size ys = n ∧ xs <_(Listn.le r) ys})")
```

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

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

```
lemma Listn_sl_aux:
assumes "semilat (A, r, f)" shows "semilat (Listn.sl n (A,r,f))"
proof -
  interpret Semilat A r f using assms by (rule Semilat.intro)
show ?thesis
apply (unfold Listn.sl_def)
apply (simp (no_asm) only: semilat_Def split_conv)
apply (rule conjI)
 apply simp
apply (rule conjI)
 apply (simp only: closedI closed_listI)
apply (simp (no_asm) only: list_def)
apply (simp (no_asm_simp) add: plus_list_ub1 plus_list_ub2 plus_list_lub)
done
qed

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

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

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

lemma coalesce_eq_OK1_D [rule_format]:
  "semilat(err A, Err.le r, lift2 f) ⟹
  !xs. xs : list n A ⟶ (!ys. ys : list n A ⟶
  (!zs. coalesce (xs +[f] ys) = OK zs ⟶ xs <=[r] zs))"
apply (induct n)
  apply simp
apply clarify
apply (simp add: in_list_Suc_iff)
apply clarify
apply (simp split: err.split_asm add: lem Err.sup_def lift2_def)
apply (force simp add: semilat_le_err_OK1)
done

lemma coalesce_eq_OK2_D [rule_format]:
  "semilat(err A, Err.le r, lift2 f) ⟹
  !xs. xs : list n A ⟶ (!ys. ys : list n A ⟶
  (!zs. coalesce (xs +[f] ys) = OK zs ⟶ ys <=[r] zs))"
apply (induct n)
 apply simp
apply clarify
```

```
apply (simp add: in_list_Suc_iff)
apply clarify
apply (simp split: err.split_asm add: lem Err.sup_def lift2_def)
apply (force simp add: semilat_le_err_OK2)
done

lemma lift2_le_ub:
  "⟦ semilat(err A, Err.le r, lift2 f); x∈ A; y∈ A; x +_f y = OK z;
     u∈ A; x <=_r u; y <=_r u ⟧ ⟹ z <=_r u"
apply (unfold semilat_Def plussub_def err_def)
apply (simp add: lift2_def)
apply clarify
apply (rotate_tac -3)
apply (erule thin_rl)
apply (erule thin_rl)
apply force
done

lemma coalesce_eq_OK_ub_D [rule_format]:
  "semilat(err A, Err.le r, lift2 f) ⟹
  !xs. xs : list n A ⟶ (!ys. ys : list n A ⟶
  (!zs us. coalesce (xs +[f] ys) = OK zs & xs <=[r] us & ys <=[r] us
          & us : list n A ⟶ zs <=[r] us))"
apply (induct n)
 apply simp
apply clarify
apply (simp add: in_list_Suc_iff)
apply clarify
apply (simp (no_asm_use) split: err.split_asm add: lem Err.sup_def lift2_def)
apply clarify
apply (rule conjI)
 apply (blast intro: lift2_le_ub)
apply blast
done

lemma lift2_eq_ErrD:
  "⟦ x +_f y = Err; semilat(err A, Err.le r, lift2 f); x∈ A; y∈ A ⟧
  ⟹ ~(∃u∈ A. x <=_r u & y <=_r u)"
  by (simp add: OK_plus_OK_eq_Err_conv [THEN iffD1])


lemma coalesce_eq_Err_D [rule_format]:
  "⟦ semilat(err A, Err.le r, lift2 f) ⟧
  ⟹ !xs. xs∈ list n A ⟶ (!ys. ys∈ list n A ⟶
     coalesce (xs +[f] ys) = Err ⟶
     ~(∃zs∈ list n A. xs <=[r] zs & ys <=[r] zs))"
apply (induct n)
 apply simp
apply clarify
apply (simp add: in_list_Suc_iff)
apply clarify
apply (simp split: err.split_asm add: lem Err.sup_def lift2_def)
 apply (blast dest: lift2_eq_ErrD)
done
```

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

lemma closed_map2_list [rule_format]:
  "closed (err A) (lift2 f) ⟹
  ∀xs. xs : list n A ⟶ (∀ys. ys : list n A ⟶
  map2 f xs ys : list n (err A))"
apply (unfold map2_def)
apply (induct n)
 apply simp
apply clarify
apply (simp add: in_list_Suc_iff)
apply clarify
apply (simp add: plussub_def closed_err_lift2_conv)
done

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

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

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

end
```

4.4 Typing and Dataflow Analysis Framework

```
theory Typing_Framework
imports Listn
begin
```

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

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

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

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

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

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

end
```

4.5 Products as Semilattices

```
theory Product
imports Err
begin

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

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

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

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

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

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

lemma less_prod_Pair_conv:
  "((a1,b1) <_(Product.le rA rB) (a2,b2)) =
  (a1 <_rA a2 & b1 <=_rB b2 | a1 <=_rA a2 & b1 <_rB b2)"
apply (unfold lesssub_def)
apply simp
apply blast
done

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

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

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

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

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


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

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

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

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

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

end
```

4.6 More on Semilattices

```
theory SemilatAlg
imports Typing_Framework Product
begin

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

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

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

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

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

lemma pres_typeD:
  "⟦ pres_type step n A; s∈A; p<n; (q,s')∈set (step p s) ⟧ ⟹ s' ∈ A"
  by (unfold pres_type_def, blast)

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

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

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

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


lemma list_update_le_listI [rule_format]:
  "set xs <= A ⟶ set ys <= A ⟶ xs <=[r] ys ⟶ p < size xs ⟶
   x <=_r ys!p ⟶ semilat(A,r,f) ⟶ x∈A ⟶
   xs[p := x +_f xs!p] <=[r] ys"
  apply (unfold Listn.le_def lesub_def semilat_def)
  apply (simp add: list_all2_conv_all_nth nth_list_update)
  done
```

```
lemma plusplus_closed: assumes "semilat (A, r, f)" shows
  "⋀y. ⟦ set x ⊆ A; y ∈ A⟧ ⟹ x ++_f y ∈ A" (is "PROP ?P")
proof -
  interpret Semilat A r f using assms by (rule Semilat.intro)
  show "PROP ?P" proof (induct x)
    show "⋀y. y ∈ A ⟹ [] ++_f y ∈ A" by simp
    fix y x xs
    assume y: "y ∈ A" and xs: "set (x#xs) ⊆ A"
    assume IH: "⋀y. ⟦ set xs ⊆ A; y ∈ A⟧ ⟹ xs ++_f y ∈ A"
    from xs obtain x: "x ∈ A" and xs': "set xs ⊆ A" by simp
    from x y have "(x +_f y) ∈ A" ..
    with xs' have "xs ++_f (x +_f y) ∈ A" by (rule IH)
    thus "(x#xs) ++_f y ∈ A" by simp
  qed
qed

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

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

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


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

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

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

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

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


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


lemma ub1':
  assumes "semilat (A, r, f)"
  shows "⟦∀(p,s) ∈ set S. s ∈ A; y ∈ A; (a,b) ∈ set S⟧
  ⟹ b <=_r map snd [(p', t')←S. p' = a] ++_f y"
proof -
  interpret Semilat A r f using assms by (rule Semilat.intro)

  let "b <=_r ?map ++_f y" = ?thesis

  assume "y ∈ A"
  moreover
```

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


lemma plusplus_empty:
  "∀s'. (q, s') ∈ set S ⟶ s' +_f ss ! q = ss ! q ⟹
   (map snd [(p', t') ← S. p' = q] ++_f ss ! q) = ss ! q"
  by (induct S) auto

end
```

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

```
theory Typing_Framework_err
imports Typing_Framework SemilatAlg
begin

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

definition wt_app_eff :: "'s ord ⇒ (nat ⇒ 's ⇒ bool) ⇒ 's step_type ⇒ 's list ⇒ bool"
where
"wt_app_eff r app step ts ≡
  ∀p < size ts. app p (ts!p) ∧ (∀(q,t) ∈ set (step p (ts!p)). t <=_r ts!q)"

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

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

definition err_step :: "nat ⇒ (nat ⇒ 's ⇒ bool) ⇒ 's step_type ⇒ 's err step_type"
where
"err_step n app step p t ≡
  case t of
    Err   ⇒ error n
  | OK t' ⇒ if app p t' then map_snd OK (step p t') else error n"

definition app_mono :: "'s ord ⇒ (nat ⇒ 's ⇒ bool) ⇒ nat ⇒ 's set ⇒ bool" where
"app_mono r app n A ≡
 ∀s p t. s ∈ A ∧ p < n ∧ s <=_r t ⟶ app p t ⟶ app p s"


lemmas err_step_defs = err_step_def map_snd_def error_def


lemma bounded_err_stepD:
  "bounded (err_step n app step) n ⟹
  p < n ⟹  app p a ⟹ (q,b) ∈ set (step p a) ⟹
  q < n"
  apply (simp add: bounded_def err_step_def)
  apply (erule allE, erule impE, assumption)
  apply (erule_tac x = "OK a" in allE, drule bspec)
   apply (simp add: map_snd_def)
   apply fast
  apply simp
  done


lemma in_map_sndD: "(a,b) ∈ set (map_snd f xs) ⟹ ∃b'. (a,b') ∈ set xs"
  apply (induct xs)
  apply (auto simp add: map_snd_def)
  done
```

```
lemma bounded_err_stepI:
  "∀p. p < n ⟶ (∀s. ap p s ⟶ (∀(q,s') ∈ set (step p s). q < n))
  ⟹ bounded (err_step n ap step) n"
apply (clarsimp simp: bounded_def err_step_def split: err.splits)
apply (simp add: error_def image_def)
apply (blast dest: in_map_sndD)
done


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


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


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


lemma mono_lift:
  "order r ⟹ app_mono r app n A ⟹ bounded (err_step n app step) n ⟹
  ∀s p t. s ∈ A ∧ p < n ∧ s <=_r t ⟶ app p t ⟶ step p s <=|r| step p t ⟹
  mono (Err.le r) (err_step n app step) n (err A)"
apply (unfold app_mono_def mono_def err_step_def)
apply clarify
apply (case_tac s)
 apply simp
apply simp
apply (case_tac t)
 apply simp
 apply clarify
 apply (simp add: lesubstep_type_def error_def)
 apply clarify
 apply (drule in_map_sndD)
 apply clarify
```

```
 apply (drule bounded_err_stepD, assumption+)
 apply (rule exI [of _ Err])
 apply simp
apply simp
apply (erule allE, erule allE, erule allE, erule impE)
 apply (rule conjI, assumption)
 apply (rule conjI, assumption)
 apply assumption
apply (rule conjI)
apply clarify
apply (erule allE, erule allE, erule allE, erule impE)
 apply (rule conjI, assumption)
 apply (rule conjI, assumption)
 apply assumption
apply (erule impE, assumption)
apply (rule map_snd_lessI, assumption)
apply clarify
apply (simp add: lesubstep_type_def error_def)
apply clarify
apply (drule in_map_sndD)
apply clarify
apply (drule bounded_err_stepD, assumption+)
apply (rule exI [of _ Err])
apply simp
done

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

lemma pres_type_lift:
  "∀s∈A. ∀p. p < n ⟶ app p s ⟶ (∀(q, s')∈set (step p s). s' ∈ A)
  ⟹ pres_type (err_step n app step) n (err A)"
apply (unfold pres_type_def err_step_def)
apply clarify
apply (case_tac b)
 apply simp
apply (case_tac s)
 apply simp
 apply (drule in_errorD)
 apply simp
apply (simp add: map_snd_def split: split_if_asm)
 apply fast
apply (drule in_errorD)
apply simp
done
```

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

```
lemma wt_err_imp_wt_app_eff:
  assumes wt: "wt_err_step r (err_step (size ts) app step) ts"
  assumes b:  "bounded (err_step (size ts) app step) (size ts)"
  shows "wt_app_eff r app step (map ok_val ts)"
```

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

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

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

  show "∀(q,t)∈set(step p (map ok_val ts ! p)). t <=_r map ok_val ts ! q"
  proof clarify
    fix q t assume q: "(q,t) ∈ set (step p (map ok_val ts ! p))"

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

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

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


lemma wt_app_eff_imp_wt_err:
  assumes app_eff: "wt_app_eff r app step ts"
  assumes bounded: "bounded (err_step (size ts) app step) (size ts)"
```

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

end
```

4.8 Kildall's Algorithm

```
theory Kildall
imports SemilatAlg While_Combinator
begin

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

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

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

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

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


lemmas [simp] = Let_def Semilat.le_iff_plus_unchanged [OF Semilat.intro, symmetric]


lemma (in Semilat) nth_merges:
 "⋀ss. ⟦p < length ss; ss ∈ list n A; ∀(p,t)∈set ps. p<n ∧ t∈A ⟧ ⟹
  (merges f ps ss)!p = map snd [(p',t') ← ps. p'=p] ++_f ss!p"
  (is "⋀ss. ⟦_; _; ?steptype ps⟧ ⟹ ?P ss ps")
proof (induct ps)
  show "⋀ss. ?P ss []" by simp

  fix ss p' ps'
  assume ss: "ss ∈ list n A"
  assume l:  "p < length ss"
  assume "?steptype (p'#ps')"
  then obtain a b where
    p': "p'=(a,b)" and ab: "a<n" "b∈A" and ps': "?steptype ps'"
    by (cases p') auto
  assume "⋀ss. p< length ss ⟹ ss ∈ list n A ⟹ ?steptype ps' ⟹ ?P ss ps'"
  from this [OF _ _ ps'] have IH: "⋀ss. ss ∈ list n A ⟹ p < length ss ⟹ ?P ss ps'"
.

  from ss ab
```

```
  have "ss[a := b +_f ss!a] ∈ list n A" by (simp add: closedD)
  moreover
  from calculation 1
  have "p < length (ss[a := b +_f ss!a])" by simp
  ultimately
  have "?P (ss[a := b +_f ss!a]) ps'" by (rule IH)
  with p' 1
  show "?P ss (p'#ps')" by simp
qed
```

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

```
lemma (in Semilat) merges_preserves_type_lemma:
shows "∀xs. xs ∈ list n A ⟶ (∀(p,x) ∈ set ps. p<n ∧ x∈A)
          ⟶ merges f ps xs ∈ list n A"
apply (insert closedI)
apply (unfold closed_def)
apply (induct_tac ps)
 apply simp
apply clarsimp
done

lemma (in Semilat) merges_preserves_type [simp]:
 "⟦ xs ∈ list n A; ∀(p,x) ∈ set ps. p<n ∧ x∈A ⟧
  ⟹ merges f ps xs ∈ list n A"
by (simp add: merges_preserves_type_lemma)

lemma (in Semilat) merges_incr_lemma:
 "∀xs. xs ∈ list n A ⟶ (∀(p,x)∈set ps. p<size xs ∧ x ∈ A) ⟶ xs <=[r] merges f ps
xs"
apply (induct_tac ps)
 apply simp
apply simp
apply clarify
apply (rule order_trans)
  apply simp
 apply (erule list_update_incr)
  apply simp
 apply simp
apply (blast intro!: listE_set intro: closedD listE_length [THEN nth_in])
done

lemma (in Semilat) merges_incr:
 "⟦ xs ∈ list n A; ∀(p,x)∈set ps. p<size xs ∧ x ∈ A ⟧
  ⟹ xs <=[r] merges f ps xs"
  by (simp add: merges_incr_lemma)
```

```
lemma (in Semilat) merges_same_conv [rule_format]:
 "(∀xs. xs ∈ list n A ⟶ (∀(p,x)∈set ps. p<size xs ∧ x∈A) ⟶
     (merges f ps xs = xs) = (∀(p,x)∈set ps. x <=_r xs!p))"
  apply (induct_tac ps)
   apply simp
  apply clarsimp
  apply (rename_tac p x ps xs)
  apply (rule iffI)
   apply (rule context_conjI)
    apply (subgoal_tac "xs[p := x +_f xs!p] <=[r] xs")
     apply (drule_tac p = p in le_listD)
      apply simp
     apply simp
    apply (erule subst, rule merges_incr)
       apply (blast intro!: listE_set intro: closedD listE_length [THEN nth_in])
      apply clarify
      apply (rule conjI)
       apply simp
       apply (blast dest: boundedD)
      apply blast
   apply clarify
   apply (erule allE)
   apply (erule impE)
    apply assumption
   apply (drule bspec)
    apply assumption
   apply (simp add: le_iff_plus_unchanged [THEN iffD1] list_update_same_conv [THEN iffD2])
   apply blast
  apply clarify
  apply (simp add: le_iff_plus_unchanged [THEN iffD1] list_update_same_conv [THEN iffD2])
  done


lemma (in Semilat) list_update_le_listI [rule_format]:
  "set xs <= A ⟶ set ys <= A ⟶ xs <=[r] ys ⟶ p < size xs ⟶
   x <=_r ys!p ⟶ x∈A ⟶ xs[p := x +_f xs!p] <=[r] ys"
  apply(insert semilat)
  apply (unfold Listn.le_def lesub_def semilat_def)
  apply (simp add: list_all2_conv_all_nth nth_list_update)
  done

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

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

  from prems show ?thesis by blast
qed
```

```
lemma decomp_propa:
  "⋀ss w. (∀(q,t)∈set qs. q < size ss) ⟹
   propa f qs ss w =
   (merges f qs ss, {q. ∃t. (q,t)∈set qs ∧ t +_f ss!q ≠ ss!q} Un w)"
  apply (induct qs)
   apply simp
  apply (simp (no_asm))
  apply clarify
  apply simp
  apply (rule conjI)
   apply blast
  apply (simp add: nth_list_update)
  apply blast
  done
```

```
lemma (in Semilat) stable_pres_lemma:
shows "⟦pres_type step n A; bounded step n;
     ss ∈ list n A; p ∈ w; ∀q∈w. q < n;
     ∀q. q < n ⟶ q ∉ w ⟶ stable r step ss q; q < n;
     ∀s'. (q,s') ∈ set (step p (ss ! p)) ⟶ s' +_f ss ! q = ss ! q;
     q ∉ w ∨ q = p ⟧
  ⟹ stable r step (merges f (step p (ss!p)) ss) q"
  apply (unfold stable_def)
  apply (subgoal_tac "∀s'. (q,s') ∈ set (step p (ss!p)) ⟶ s' : A")
   prefer 2
   apply clarify
   apply (erule pres_typeD)
    prefer 3 apply assumption
    apply (rule listE_nth_in)
     apply assumption
    apply simp
   apply simp
  apply simp
  apply clarify
```

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

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

 apply simp
 apply (erule allE, erule impE, assumption, erule impE, assumption)
 apply (rule order_trans)
   apply simp
  defer
 apply (rule pp_ub2)
   apply simp
   apply clarify
   apply simp
   apply (rule pres_typeD)
      apply assumption
     prefer 3 apply assumption
    apply (blast intro: listE_nth_in dest: boundedD)
   apply (blast intro: pres_typeD dest: boundedD)
  apply (blast intro: listE_nth_in dest: boundedD)
```

```
 apply blast
 done


lemma (in Semilat) merges_bounded_lemma:
 "⟦ mono r step n A; bounded step n;
    ∀(p',s') ∈ set (step p (ss!p)). s' ∈ A; ss ∈ list n A; ts ∈ list n A; p < n;
    ss <=[r] ts; ∀p. p < n ⟶ stable r step ts p ⟧
  ⟹ merges f (step p (ss!p)) ss <=[r] ts"
  apply (unfold stable_def)
  apply (rule merges_pres_le_ub)
     apply simp
    apply simp
   prefer 2 apply assumption

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

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

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

lemma termination_lemma:
  assumes semilat: "semilat (A, r, f)"
  shows "⟦ ss ∈ list n A; ∀(q,t)∈set qs. q<n ∧ t∈A; p∈w ⟧ ⟹
  ss <[r] merges f qs ss ∨
  merges f qs ss = ss ∧ {q. ∃t. (q,t)∈set qs ∧ t +_f ss!q ≠ ss!q} Un (w-{p}) < w" (is
"PROP ?P")
proof -
  interpret Semilat A r f using assms by (rule Semilat.intro)
  show "PROP ?P" apply(insert semilat)
    apply (unfold lesssub_def)
    apply (simp (no_asm_simp) add: merges_incr)
    apply (rule impI)
    apply (rule merges_same_conv [THEN iffD1, elim_format])
    apply assumption+
    defer
    apply (rule sym, assumption)
    defer apply simp
    apply (subgoal_tac "∀q t. ¬((q, t) ∈ set qs ∧ t +_f ss ! q ≠ ss ! q)")
    apply (blast intro!: psubsetI elim: equalityE)
    apply clarsimp
    apply (drule bspec, assumption)
```

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

lemma iter_properties[rule_format]:
  assumes semilat: "semilat (A, r, f)"
  shows "⟦ acc r ; pres_type step n A; mono r step n A;
     bounded step n; ∀p∈w0. p < n; ss0 ∈ list n A;
     ∀p<n. p ∉ w0 ⟶ stable r step ss0 p ⟧ ⟹
   iter f step ss0 w0 = (ss',w')
   ⟶
   ss' ∈ list n A ∧ stables r step ss' ∧ ss0 <=[r] ss' ∧
   (∀ts∈list n A. ss0 <=[r] ts ∧ stables r step ts ⟶ ss' <=[r] ts)"
  (is "PROP ?P")
proof -
  interpret Semilat A r f using assms by (rule Semilat.intro)
  show "PROP ?P" apply(insert semilat)
apply (unfold iter_def stables_def)
apply (rule_tac P = "%(ss,w).
 ss ∈ list n A ∧ (∀p<n. p ∉ w ⟶ stable r step ss p) ∧ ss0 <=[r] ss ∧
 (∀ts∈list n A. ss0 <=[r] ts ∧ stables r step ts ⟶ ss <=[r] ts) ∧
 (∀p∈w. p < n)" and
 r = "{(ss',ss) . ss <[r] ss'} <*lex*> finite_psubset"
      in while_rule)

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

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

```
  apply(clarsimp, fast dest!: boundedD)
 apply (erule pres_typeD)
  prefer 3
  apply assumption
  apply (erule listE_nth_in)
  apply blast
 apply blast
apply (rule conjI)
 apply clarify
 apply (blast intro!: stable_pres_lemma)
apply (rule conjI)
 apply (blast intro!: merges_incr intro: le_list_trans)
apply (rule conjI)
 apply clarsimp
 apply (blast intro!: merges_bounded_lemma)
apply (blast dest!: boundedD)
```

— Postcondition holds upon termination:

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

— Well-foundedness of the termination relation:

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

— Loop decreases along termination relation:

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

```
apply clarsimp
done

qed

lemma kildall_properties:
assumes semilat: "semilat (A, r, f)"
shows "⟦ acc r; pres_type step n A; mono r step n A;
     bounded step n; ss0 ∈ list n A ⟧ ⟹
  kildall r f step ss0 ∈ list n A ∧
  stables r step (kildall r f step ss0) ∧
  ss0 <=[r] kildall r f step ss0 ∧
  (∀ts∈list n A. ss0 <=[r] ts ∧ stables r step ts ⟶
                 kildall r f step ss0 <=[r] ts)"
  (is "PROP ?P")
proof -
  interpret Semilat A r f using assms by (rule Semilat.intro)
  show "PROP ?P"
apply (unfold kildall_def)
apply(case_tac "iter f step ss0 (unstables r step ss0)")
apply(simp)
apply (rule iter_properties)
apply (simp_all add: unstables_def stable_def)
apply (rule semilat)
done
qed

lemma is_bcv_kildall:
assumes semilat: "semilat (A, r, f)"
shows "⟦ acc r; top r T; pres_type step n A; bounded step n; mono r step n A ⟧
  ⟹ is_bcv r T step n A (kildall r f step)"
  (is "PROP ?P")
proof -
  interpret Semilat A r f using assms by (rule Semilat.intro)
  show "PROP ?P"
apply(unfold is_bcv_def wt_step_def)
apply(insert semilat kildall_properties[of A])
apply(simp add:stables_def)
apply clarify
apply(subgoal_tac "kildall r f step ss ∈ list n A")
 prefer 2 apply (simp(no_asm_simp))
apply (rule iffI)
 apply (rule_tac x = "kildall r f step ss" in bexI)
  apply (rule conjI)
   apply (blast)
  apply (simp  (no_asm_simp))
 apply(assumption)
apply clarify
apply(subgoal_tac "kildall r f step ss!p <=_r ts!p")
 apply simp
apply (blast intro!: le_listD less_lengthI)
done
qed
```

end

4.9 More about Options

```
theory Opt
imports Err
begin

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

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

definition sup :: "'a ebinop ⇒ 'a option ebinop" where
"sup f o1 o2 ==
 case o1 of None ⇒ OK o2 | Some x ⇒ (case o2 of None ⇒ OK o1
     | Some y ⇒ (case f x y of Err ⇒ Err | OK z ⇒ OK (Some z)))"

definition esl :: "'a esl ⇒ 'a option esl" where
"esl == %(A,r,f). (opt A, le r, sup f)"

lemma unfold_le_opt:
  "o1 <=_(le r) o2 =
  (case o2 of None ⇒ o1=None |
              Some y ⇒ (case o1 of None ⇒ True | Some x ⇒ x <=_r y))"
apply (unfold lesub_def le_def)
apply (rule refl)
done

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

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

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

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

lemma None_bot [iff]:
```

```
  "None <=_(le r) ox"
apply (unfold lesub_def le_def)
apply (simp split: option.split)
done

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

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


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

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

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


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

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


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

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

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

  from s
```

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

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

from ord
have "order ?r"
  by simp

moreover

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

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

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

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

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

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

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

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

moreover

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

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

moreover

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

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

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

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

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

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

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

  ultimately

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

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

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


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

```
 apply (erule_tac x = "{a . Some a : Q}" in allE)
 apply blast
apply (case_tac "x")
 apply blast
apply blast
done

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

end
```

4.10 The Lightweight Bytecode Verifier

theory *LBVSpec*
imports *SemilatAlg Opt*
begin

types
  's certificate = "'s list"

primrec *merge :: "'s certificate ⇒ 's binop ⇒ 's ord ⇒ 's ⇒ nat ⇒ (nat × 's) list ⇒ 's ⇒ 's"* **where**

```
  "merge cert f r T pc []     x = x"
| "merge cert f r T pc (s#ss) x = merge cert f r T pc ss (let (pc',s') = s in
                                  if pc'=pc+1 then s' +_f x
                                  else if s' <=_r (cert!pc') then x
                                  else T)"
```

definition *wtl_inst :: "'s certificate ⇒ 's binop ⇒ 's ord ⇒ 's ⇒ 's step_type ⇒ nat ⇒ 's ⇒ 's"* **where**

```
"wtl_inst cert f r T step pc s ≡ merge cert f r T pc (step pc s) (cert!(pc+1))"
```

definition *wtl_cert :: "'s certificate ⇒ 's binop ⇒ 's ord ⇒ 's ⇒ 's ⇒ 's step_type ⇒ nat ⇒ 's ⇒ 's"* **where**

```
"wtl_cert cert f r T B step pc s ≡
  if cert!pc = B then
    wtl_inst cert f r T step pc s
  else
    if s <=_r (cert!pc) then wtl_inst cert f r T step pc (cert!pc) else T"
```

primrec *wtl_inst_list :: "'a list ⇒ 's certificate ⇒ 's binop ⇒ 's ord ⇒ 's ⇒ 's ⇒ 's step_type ⇒ nat ⇒ 's ⇒ 's"* **where**

```
  "wtl_inst_list []     cert f r T B step pc s = s"
| "wtl_inst_list (i#is) cert f r T B step pc s =
    (let s' = wtl_cert cert f r T B step pc s in
      if s' = T ∨ s = T then T else wtl_inst_list is cert f r T B step (pc+1) s')"
```

definition *cert_ok :: "'s certificate ⇒ nat ⇒ 's ⇒ 's ⇒ 's set ⇒ bool"* **where**

```
  "cert_ok cert n T B A ≡ (∀i < n. cert!i ∈ A ∧ cert!i ≠ T) ∧ (cert!n = B)"
```

definition *bottom :: "'a ord ⇒ 'a ⇒ bool"* **where**

```
  "bottom r B ≡ ∀x. B <=_r x"
```

locale *lbv = Semilat +*
  fixes *T :: "'a" ("⊤")*
  fixes *B :: "'a" ("⊥")*
  fixes *step :: "'a step_type"*
  assumes *top: "top r ⊤"*
  assumes *T_A: "⊤ ∈ A"*
  assumes *bot: "bottom r ⊥"*
  assumes *B_A: "⊥ ∈ A"*

  fixes *merge :: "'a certificate ⇒ nat ⇒ (nat × 'a) list ⇒ 'a ⇒ 'a"*
  defines *mrg_def: "merge cert ≡ LBVSpec.merge cert f r ⊤"*

```
  fixes wti :: "'a certificate ⇒ nat ⇒ 'a ⇒ 'a"
  defines wti_def: "wti cert ≡ wtl_inst cert f r ⊤ step"

  fixes wtc :: "'a certificate ⇒ nat ⇒ 'a ⇒ 'a"
  defines wtc_def: "wtc cert ≡ wtl_cert cert f r ⊤ ⊥ step"

  fixes wtl :: "'b list ⇒ 'a certificate ⇒ nat ⇒ 'a ⇒ 'a"
  defines wtl_def: "wtl ins cert ≡ wtl_inst_list ins cert f r ⊤ ⊥ step"


lemma (in lbv) wti:
  "wti c pc s ≡ merge c pc (step pc s) (c!(pc+1))"
  by (simp add: wti_def mrg_def wtl_inst_def)

lemma (in lbv) wtc:
  "wtc c pc s ≡ if c!pc = ⊥ then wti c pc s else if s <=_r c!pc then wti c pc (c!pc)
else ⊤"
  by (unfold wtc_def wti_def wtl_cert_def)


lemma cert_okD1 [intro?]:
  "cert_ok c n T B A ⟹ pc < n ⟹ c!pc ∈ A"
  by (unfold cert_ok_def) fast

lemma cert_okD2 [intro?]:
  "cert_ok c n T B A ⟹ c!n = B"
  by (simp add: cert_ok_def)

lemma cert_okD3 [intro?]:
  "cert_ok c n T B A ⟹ B ∈ A ⟹ pc < n ⟹ c!Suc pc ∈ A"
  by (drule Suc_leI) (auto simp add: le_eq_less_or_eq dest: cert_okD1 cert_okD2)

lemma cert_okD4 [intro?]:
  "cert_ok c n T B A ⟹ pc < n ⟹ c!pc ≠ T"
  by (simp add: cert_ok_def)

declare Let_def [simp]
```

4.10.1 more semilattice lemmas

```
lemma (in lbv) sup_top [simp, elim]:
  assumes x: "x ∈ A"
  shows "x +_f ⊤ = ⊤"
proof -
  from top have "x +_f ⊤ <=_r ⊤" ..
  moreover from x T_A have "⊤ <=_r x +_f ⊤" ..
  ultimately show ?thesis ..
qed

lemma (in lbv) plusplussup_top [simp, elim]:
  "set xs ⊆ A ⟹ xs ++_f ⊤ = ⊤"
  by (induct xs) auto
```

```
lemma (in Semilat) pp_ub1':
  assumes S: "snd`set S ⊆ A"
  assumes y: "y ∈ A" and ab: "(a, b) ∈ set S"
  shows "b <=_r map snd [(p', t') ← S . p' = a] ++_f y"
proof -
  from S have "∀(x,y) ∈ set S. y ∈ A" by auto
  with semilat y ab show ?thesis by - (rule ub1')
qed

lemma (in lbv) bottom_le [simp, intro]:
  "⊥ <=_r x"
  by (insert bot) (simp add: bottom_def)

lemma (in lbv) le_bottom [simp]:
  "x <=_r ⊥ = (x = ⊥)"
  by (blast intro: antisym_r)
```

4.10.2 merge

```
lemma (in lbv) merge_Nil [simp]:
  "merge c pc [] x = x" by (simp add: mrg_def)

lemma (in lbv) merge_Cons [simp]:
  "merge c pc (l#ls) x = merge c pc ls (if fst l=pc+1 then snd l +_f x
                                        else if snd l <=_r (c!fst l) then x
                                        else ⊤)"
  by (simp add: mrg_def split_beta)

lemma (in lbv) merge_Err [simp]:
  "snd`set ss ⊆ A ⟹ merge c pc ss ⊤ = ⊤"
  by (induct ss) auto

lemma (in lbv) merge_not_top:
  "⋀x. snd`set ss ⊆ A ⟹ merge c pc ss x ≠ ⊤ ⟹
  ∀(pc',s') ∈ set ss. (pc' ≠ pc+1 ⟶ s' <=_r (c!pc'))"
  (is "⋀x. ?set ss ⟹ ?merge ss x ⟹ ?P ss")
proof (induct ss)
  show "?P []" by simp
next
  fix x ls l
  assume "?set (l#ls)" then obtain set: "snd`set ls ⊆ A" by simp
  assume merge: "?merge (l#ls) x"
  moreover
  obtain pc' s' where l: "l = (pc',s')" by (cases l)
  ultimately
  obtain x' where merge': "?merge ls x'" by simp
  assume "⋀x. ?set ls ⟹ ?merge ls x ⟹ ?P ls" hence "?P ls" using set merge' .
  moreover
  from merge set
  have "pc' ≠ pc+1 ⟶ s' <=_r (c!pc')" by (simp add: l split: split_if_asm)
  ultimately
  show "?P (l#ls)" by (simp add: l)
```

```
qed


lemma (in lbv) merge_def:
  shows
  "⋀x. x ∈ A ⟹ snd`set ss ⊆ A ⟹
  merge c pc ss x =
  (if ∀(pc',s') ∈ set ss. pc'≠pc+1 ⟶ s' <=_r c!pc' then
    map snd [(p',t') ← ss. p'=pc+1] ++_f x
  else ⊤)"
  (is "⋀x. _ ⟹ _ ⟹ ?merge ss x = ?if ss x" is "⋀x. _ ⟹ _ ⟹ ?P ss x")
proof (induct ss)
  fix x show "?P [] x" by simp
next
  fix x assume x: "x ∈ A"
  fix l::"nat × 'a" and ls
  assume "snd`set (l#ls) ⊆ A"
  then obtain l: "snd l ∈ A" and ls: "snd`set ls ⊆ A" by auto
  assume "⋀x. x ∈ A ⟹ snd`set ls ⊆ A ⟹ ?P ls x"
  hence IH: "⋀x. x ∈ A ⟹ ?P ls x" using ls by iprover
  obtain pc' s' where [simp]: "l = (pc',s')" by (cases l)
  hence "?merge (l#ls) x = ?merge ls
    (if pc'=pc+1 then s' +_f x else if s' <=_r c!pc' then x else ⊤)"
    (is "?merge (l#ls) x = ?merge ls ?if'")
    by simp
  also have "... = ?if ls ?if'"
  proof -
    from l have "s' ∈ A" by simp
    with x have "s' +_f x ∈ A" by simp
    with x T_A have "?if' ∈ A" by auto
    hence "?P ls ?if'" by (rule IH) thus ?thesis by simp
  qed
  also have "... = ?if (l#ls) x"
    proof (cases "∀(pc', s')∈set (l#ls). pc'≠pc+1 ⟶ s' <=_r c!pc'")
      case True
      hence "∀(pc', s')∈set ls. pc'≠pc+1 ⟶ s' <=_r c!pc'" by auto
      moreover
      from True have
        "map snd [(p',t')←ls . p'=pc+1] ++_f ?if' =
        (map snd [(p',t')←l#ls . p'=pc+1] ++_f x)"
        by simp
      ultimately
      show ?thesis using True by simp
    next
      case False
      moreover
      from ls have "set (map snd [(p', t')←ls . p' = Suc pc]) ⊆ A" by auto
      ultimately show ?thesis by auto
    qed
  finally show "?P (l#ls) x" .
qed

lemma (in lbv) merge_not_top_s:
  assumes x:  "x ∈ A" and ss: "snd`set ss ⊆ A"
```

```
  assumes m:  "merge c pc ss x ≠ ⊤"
  shows "merge c pc ss x = (map snd [(p',t') ← ss. p'=pc+1] ++_f x)"
proof -
  from ss m have "∀(pc',s') ∈ set ss. (pc' ≠ pc+1 ⟶ s' <=_r c!pc')"
    by (rule merge_not_top)
  with x ss m show ?thesis by - (drule merge_def, auto split: split_if_asm)
qed
```

4.10.3 wtl-inst-list

```
lemmas [iff] = not_Err_eq

lemma (in lbv) wtl_Nil [simp]: "wtl [] c pc s = s"
  by (simp add: wtl_def)

lemma (in lbv) wtl_Cons [simp]:
  "wtl (i#is) c pc s =
  (let s' = wtc c pc s in if s' = ⊤ ∨ s = ⊤ then ⊤ else wtl is c (pc+1) s')"
  by (simp add: wtl_def wtc_def)

lemma (in lbv) wtl_Cons_not_top:
  "wtl (i#is) c pc s ≠ ⊤ =
  (wtc c pc s ≠ ⊤ ∧ s ≠ T ∧ wtl is c (pc+1) (wtc c pc s) ≠ ⊤)"
  by (auto simp del: split_paired_Ex)

lemma (in lbv) wtl_top [simp]:  "wtl ls c pc ⊤ = ⊤"
  by (cases ls) auto

lemma (in lbv) wtl_not_top:
  "wtl ls c pc s ≠ ⊤ ⟹ s ≠ ⊤"
  by (cases "s=⊤") auto

lemma (in lbv) wtl_append [simp]:
  "⋀pc s. wtl (a@b) c pc s = wtl b c (pc+length a) (wtl a c pc s)"
  by (induct a) auto

lemma (in lbv) wtl_take:
  "wtl is c pc s ≠ ⊤ ⟹ wtl (take pc' is) c pc s ≠ ⊤"
  (is "?wtl is ≠ _ ⟹ _")
proof -
  assume "?wtl is ≠ ⊤"
  hence "?wtl (take pc' is @ drop pc' is) ≠ ⊤" by simp
  thus ?thesis by (auto dest!: wtl_not_top simp del: append_take_drop_id)
qed

lemma take_Suc:
  "∀n. n < length l ⟶ take (Suc n) l = (take n l)@[l!n]" (is "?P l")
proof (induct l)
  show "?P []" by simp
next
  fix x xs assume IH: "?P xs"
  show "?P (x#xs)"
  proof (intro strip)
    fix n assume "n < length (x#xs)"
```

```
    with IH show "take (Suc n) (x # xs) = take n (x # xs) @ [(x # xs) ! n]"
      by (cases n, auto)
  qed
qed

lemma (in lbv) wtl_Suc:
  assumes suc: "pc+1 < length is"
  assumes wtl: "wtl (take pc is) c 0 s ≠ ⊤"
  shows "wtl (take (pc+1) is) c 0 s = wtc c pc (wtl (take pc is) c 0 s)"
proof -
  from suc have "take (pc+1) is=(take pc is)@[is!pc]" by (simp add: take_Suc)
  with suc wtl show ?thesis by (simp add: min_max.inf_absorb2)
qed

lemma (in lbv) wtl_all:
  assumes all: "wtl is c 0 s ≠ ⊤" (is "?wtl is ≠ _")
  assumes pc:  "pc < length is"
  shows  "wtc c pc (wtl (take pc is) c 0 s) ≠ ⊤"
proof -
  from pc have "0 < length (drop pc is)" by simp
  then  obtain i r where Cons: "drop pc is = i#r"
    by (auto simp add: neq_Nil_conv simp del: length_drop drop_eq_Nil)
  hence "i#r = drop pc is" ..
  with all have take: "?wtl (take pc is@i#r) ≠ ⊤" by simp
  from pc have "is!pc = drop pc is ! 0" by simp
  with Cons have "is!pc = i" by simp
  with take pc show ?thesis by (auto simp add: min_max.inf_absorb2)
qed
```

4.10.4 preserves-type

```
lemma (in lbv) merge_pres:
  assumes s0: "snd‘set ss ⊆ A" and x: "x ∈ A"
  shows "merge c pc ss x ∈ A"
proof -
  from s0 have "set (map snd [(p', t')←ss . p'=pc+1]) ⊆ A" by auto
  with x  have "(map snd [(p', t')←ss . p'=pc+1] ++_f x) ∈ A"
    by (auto intro!: plusplus_closed semilat)
  with s0 x show ?thesis by (simp add: merge_def T_A)
qed


lemma pres_typeD2:
  "pres_type step n A ⟹ s ∈ A ⟹ p < n ⟹ snd‘set (step p s) ⊆ A"
  by auto (drule pres_typeD)


lemma (in lbv) wti_pres [intro?]:
  assumes pres: "pres_type step n A"
  assumes cert: "c!(pc+1) ∈ A"
  assumes s_pc: "s ∈ A" "pc < n"
  shows "wti c pc s ∈ A"
proof -
  from pres s_pc have "snd‘set (step pc s) ⊆ A" by (rule pres_typeD2)
```

```
  with cert show ?thesis by (simp add: wti merge_pres)
qed


lemma (in lbv) wtc_pres:
  assumes pres: "pres_type step n A"
  assumes cert: "c!pc ∈ A" and cert': "c!(pc+1) ∈ A"
  assumes s: "s ∈ A" and pc: "pc < n"
  shows "wtc c pc s ∈ A"
proof -
  have "wti c pc s ∈ A" using pres cert' s pc ..
  moreover have "wti c pc (c!pc) ∈ A" using pres cert' cert pc ..
  ultimately show ?thesis using T_A by (simp add: wtc)
qed


lemma (in lbv) wtl_pres:
  assumes pres: "pres_type step (length is) A"
  assumes cert: "cert_ok c (length is) ⊤ ⊥ A"
  assumes s:    "s ∈ A"
  assumes all:  "wtl is c 0 s ≠ ⊤"
  shows "pc < length is ⟹ wtl (take pc is) c 0 s ∈ A"
  (is "?len pc ⟹ ?wtl pc ∈ A")
proof (induct pc)
  from s show "?wtl 0 ∈ A" by simp
next
  fix n assume IH: "Suc n < length is"
  then have n: "n < length is" by simp
  from IH have n1: "n+1 < length is" by simp
  assume prem: "n < length is ⟹ ?wtl n ∈ A"
  have "wtc c n (?wtl n) ∈ A"
  using pres _ _ _ n
  proof (rule wtc_pres)
    from prem n show "?wtl n ∈ A" .
    from cert n show "c!n ∈ A" by (rule cert_okD1)
    from cert n1 show "c!(n+1) ∈ A" by (rule cert_okD1)
  qed
  also
  from all n have "?wtl n ≠ ⊤" by - (rule wtl_take)
  with n1 have "wtc c n (?wtl n) = ?wtl (n+1)" by (rule wtl_Suc [symmetric])
  finally  show "?wtl (Suc n) ∈ A" by simp
qed

end
```

4.11 Correctness of the LBV

```
theory LBVCorrect
imports LBVSpec Typing_Framework
begin

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

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


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

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

lemma (in lbvs) phi_len [simp]:
  "length φ = length ins"
  by (simp add: phi_def)


lemma (in lbvs) wtl_suc_pc:
  assumes all: "wtl ins c 0 s₀ ≠ ⊤"
  assumes pc:  "pc+1 < length ins"
  shows "wtl (take (pc+1) ins) c 0 s0 ⊑r φ!(pc+1)"
proof -
  from all pc
  have "wtc c (pc+1) (wtl (take (pc+1) ins) c 0 s0) ≠ T" by (rule wtl_all)
  with pc show ?thesis by (simp add: phi_def wtc split: split_if_asm)
qed


lemma (in lbvs) wtl_stable:
  assumes wtl: "wtl ins c 0 s0 ≠ ⊤"
  assumes s0:  "s0 ∈ A"
  assumes pc:  "pc < length ins"
  shows "stable r step φ pc"
proof (unfold stable_def, clarify)
  fix pc' s' assume step: "(pc',s') ∈ set (step pc (φ ! pc))"
                    (is "(pc',s') ∈ set (?step pc)")

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

```
from wtl have tkpc: "wtl (take pc ins) c 0 s0 ≠ ⊤" (is "?s1 ≠ _") by (rule wtl_take)
from wtl have s2: "wtl (take (pc+1) ins) c 0 s0 ≠ ⊤" (is "?s2 ≠ _") by (rule wtl_take)

from wtl pc have wt_s1: "wtc c pc ?s1 ≠ ⊤" by (rule wtl_all)

have c_Some: "∀pc t. pc < length ins ⟶ c!pc ≠ ⊥ ⟶ φ!pc = c!pc"
  by (simp add: phi_def)
from pc have c_None: "c!pc = ⊥ ⟹ φ!pc = ?s1" ..

from wt_s1 pc c_None c_Some
have inst: "wtc c pc ?s1  = wti c pc (φ!pc)"
  by (simp add: wtc split: split_if_asm)

from pres cert s0 wtl pc have "?s1 ∈ A" by (rule wtl_pres)
with pc c_Some cert c_None
have "φ!pc ∈ A" by (cases "c!pc = ⊥") (auto dest: cert_okD1)
with pc pres
have step_in_A: "snd`set (?step pc) ⊆ A" by (auto dest: pres_typeD2)

show "s' <=_r φ!pc'"
proof (cases "pc' = pc+1")
  case True
  with pc' cert
  have cert_in_A: "c!(pc+1) ∈ A" by (auto dest: cert_okD1)
  from True pc' have pc1: "pc+1 < length ins" by simp
  with tkpc have "?s2 = wtc c pc ?s1" by - (rule wtl_Suc)
  with inst
  have merge: "?s2 = merge c pc (?step pc) (c!(pc+1))" by (simp add: wti)
  also
  from s2 merge have "... ≠ ⊤" (is "?merge ≠ _") by simp
  with cert_in_A step_in_A
  have "?merge = (map snd [(p',t') ← ?step pc. p'=pc+1] ++_f (c!(pc+1)))"
    by (rule merge_not_top_s)
  finally
  have "s' <=_r ?s2" using step_in_A cert_in_A True step
    by (auto intro: pp_ub1')
  also
  from wtl pc1 have "?s2 <=_r φ!(pc+1)" by (rule wtl_suc_pc)
  also note True [symmetric]
  finally show ?thesis by simp
next
  case False
  from wt_s1 inst
  have "merge c pc (?step pc) (c!(pc+1)) ≠ ⊤" by (simp add: wti)
  with step_in_A
  have "∀(pc', s')∈set (?step pc). pc'≠pc+1 ⟶ s' <=_r c!pc'"
    by - (rule merge_not_top)
  with step False
  have ok: "s' <=_r c!pc'" by blast
  moreover
  from ok
  have "c!pc' = ⊥ ⟹ s' = ⊥" by simp
  moreover
```

```
    from c_Some pc'
    have "c!pc' ≠ ⊥ ⟹ φ!pc' = c!pc'" by auto
    ultimately
    show ?thesis by (cases "c!pc' = ⊥") auto
  qed
qed


lemma (in lbvs) phi_not_top:
  assumes wtl: "wtl ins c 0 s0 ≠ ⊤"
  assumes pc:  "pc < length ins"
  shows "φ!pc ≠ ⊤"
proof (cases "c!pc = ⊥")
  case False with pc
  have "φ!pc = c!pc" ..
  also from cert pc have "... ≠ ⊤" by (rule cert_okD4)
  finally show ?thesis .
next
  case True with pc
  have "φ!pc = wtl (take pc ins) c 0 s0" ..
  also from wtl have "... ≠ ⊤" by (rule wtl_take)
  finally show ?thesis .
qed

lemma (in lbvs) phi_in_A:
  assumes wtl: "wtl ins c 0 s0 ≠ ⊤"
  assumes s0:  "s0 ∈ A"
  shows "φ ∈ list (length ins) A"
proof -
  { fix x assume "x ∈ set φ"
    then obtain xs ys where "φ = xs @ x # ys"
      by (auto simp add: in_set_conv_decomp)
    then obtain pc where pc: "pc < length φ" and x: "φ!pc = x"
      by (simp add: that [of "length xs"] nth_append)

    from pres cert wtl s0 pc
    have "wtl (take pc ins) c 0 s0 ∈ A" by (auto intro!: wtl_pres)
    moreover
    from pc have "pc < length ins" by simp
    with cert have "c!pc ∈ A" ..
    ultimately
    have "φ!pc ∈ A" using pc by (simp add: phi_def)
    hence "x ∈ A" using x by simp
  }
  hence "set φ ⊆ A" ..
  thus ?thesis by (unfold list_def) simp
qed


lemma (in lbvs) phi0:
  assumes wtl: "wtl ins c 0 s0 ≠ ⊤"
  assumes 0:   "0 < length ins"
  shows "s0 <=_r φ!0"
proof (cases "c!0 = ⊥")
```

```
  case True
  with 0 have "φ!0 = wtl (take 0 ins) c 0 s0" ..
  moreover have "wtl (take 0 ins) c 0 s0 = s0" by simp
  ultimately have "φ!0 = s0" by simp
  thus ?thesis by simp
next
  case False
  with 0 have "phi!0 = c!0" ..
  moreover
  from wtl have "wtl (take 1 ins) c 0 s0 ≠ ⊤"  by (rule wtl_take)
  with 0 False
  have "s0 <=_r c!0" by (auto simp add: neq_Nil_conv wtc split: split_if_asm)
  ultimately
  show ?thesis by simp
qed


theorem (in lbvs) wtl_sound:
  assumes wtl: "wtl ins c 0 s0 ≠ ⊤"
  assumes s0: "s0 ∈ A"
  shows "∃ts. wt_step r ⊤ step ts"
proof -
  have "wt_step r ⊤ step φ"
  proof (unfold wt_step_def, intro strip conjI)
    fix pc assume "pc < length φ"
    then have pc: "pc < length ins" by simp
    with wtl show "φ!pc ≠ ⊤" by (rule phi_not_top)
    from wtl s0 pc show "stable r step φ pc" by (rule wtl_stable)
  qed
  thus ?thesis ..
qed


theorem (in lbvs) wtl_sound_strong:
  assumes wtl: "wtl ins c 0 s0 ≠ ⊤"
  assumes s0: "s0 ∈ A"
  assumes nz: "0 < length ins"
  shows "∃ts ∈ list (length ins) A. wt_step r ⊤ step ts ∧ s0 <=_r ts!0"
proof -
  from wtl s0 have "φ ∈ list (length ins) A" by (rule phi_in_A)
  moreover
  have "wt_step r ⊤ step φ"
  proof (unfold wt_step_def, intro strip conjI)
    fix pc assume "pc < length φ"
    then have pc: "pc < length ins" by simp
    with wtl show "φ!pc ≠ ⊤" by (rule phi_not_top)
    from wtl s0 pc show "stable r step φ pc" by (rule wtl_stable)
  qed
  moreover
  from wtl nz have "s0 <=_r φ!0" by (rule phi0)
  ultimately
  show ?thesis by fast
qed
```

end

4.12 Completeness of the LBV

```
theory LBVComplete
imports LBVSpec Typing_Framework
begin

definition is_target :: "['s step_type, 's list, nat] ⇒ bool" where
  "is_target step phi pc' ≡
     ∃pc s'. pc' ≠ pc+1 ∧ pc < length phi ∧ (pc',s') ∈ set (step pc (phi!pc))"

definition make_cert :: "['s step_type, 's list, 's] ⇒ 's certificate" where
  "make_cert step phi B ≡
     map (λpc. if is_target step phi pc then phi!pc else B) [0..<length phi] @ [B]"

lemma [code]:
  "is_target step phi pc' =
  list_ex (λpc. pc' ≠ pc+1 ∧ pc' mem (map fst (step pc (phi!pc)))) [0..<length phi]"
by (force simp: list_ex_iff is_target_def mem_iff)


locale lbvc = lbv +
  fixes phi :: "'a list" ("φ")
  fixes c   :: "'a list"
  defines cert_def: "c ≡ make_cert step φ ⊥"

  assumes mono: "mono r step (length φ) A"
  assumes pres: "pres_type step (length φ) A"
  assumes phi:  "∀pc < length φ. φ!pc ∈ A ∧ φ!pc ≠ ⊤"
  assumes bounded: "bounded step (length φ)"

  assumes B_neq_T: "⊥ ≠ ⊤"


lemma (in lbvc) cert: "cert_ok c (length φ) ⊤ ⊥ A"
proof (unfold cert_ok_def, intro strip conjI)
  note [simp] = make_cert_def cert_def nth_append

  show "c!length φ = ⊥" by simp

  fix pc assume pc: "pc < length φ"
  from pc phi B_A show "c!pc ∈ A" by simp
  from pc phi B_neq_T show "c!pc ≠ ⊤" by simp
qed

lemmas [simp del] = split_paired_Ex


lemma (in lbvc) cert_target [intro?]:
  "⟦ (pc',s') ∈ set (step pc (φ!pc));
      pc' ≠ pc+1; pc < length φ; pc' < length φ ⟧
  ⟹ c!pc' = φ!pc'"
  by (auto simp add: cert_def make_cert_def nth_append is_target_def)
```

```
lemma (in lbvc) cert_approx [intro?]:
  "⟦ pc < length φ; c!pc ≠ ⊥ ⟧
  ⟹ c!pc = φ!pc"
  by (auto simp add: cert_def make_cert_def nth_append)


lemma (in lbv) le_top [simp, intro]:
  "x <=_r ⊤"
  by (insert top) simp


lemma (in lbv) merge_mono:
  assumes less:  "ss2 <=|r| ss1"
  assumes x:     "x ∈ A"
  assumes ss1:   "snd‘set ss1 ⊆ A"
  assumes ss2:   "snd‘set ss2 ⊆ A"
  shows "merge c pc ss2 x <=_r merge c pc ss1 x" (is "?s2 <=_r ?s1")
proof-
  have "?s1 = ⊤ ⟹ ?thesis" by simp
  moreover {
    assume merge: "?s1 ≠ T"
    from x ss1 have "?s1 =
      (if ∀(pc', s')∈set ss1. pc' ≠ pc + 1 ⟶ s' <=_r c!pc'
      then (map snd [(p', t') ← ss1 . p'=pc+1]) ++_f x
      else ⊤)"
      by (rule merge_def)
    with merge obtain
      app: "∀(pc',s')∈set ss1. pc' ≠ pc+1 ⟶ s' <=_r c!pc'"
           (is "?app ss1") and
      sum: "(map snd [(p',t') ← ss1 . p' = pc+1] ++_f x) = ?s1"
           (is "?map ss1 ++_f x = _" is "?sum ss1 = _")
      by (simp split: split_if_asm)
    from app less
    have "?app ss2" by (blast dest: trans_r lesub_step_typeD)
    moreover {
      from ss1 have map1: "set (?map ss1) ⊆ A" by auto
      with x have "?sum ss1 ∈ A" by (auto intro!: plusplus_closed semilat)
      with sum have "?s1 ∈ A" by simp
      moreover
      have mapD: "⋀x ss. x ∈ set (?map ss) ⟹ ∃p. (p,x) ∈ set ss ∧ p=pc+1" by auto
      from x map1
      have "∀x ∈ set (?map ss1). x <=_r ?sum ss1"
        by clarify (rule pp_ub1)
      with sum have "∀x ∈ set (?map ss1). x <=_r ?s1" by simp
      with less have "∀x ∈ set (?map ss2). x <=_r ?s1"
        by (fastsimp dest!: mapD lesub_step_typeD intro: trans_r)
      moreover
      from map1 x have "x <=_r (?sum ss1)" by (rule pp_ub2)
      with sum have "x <=_r ?s1" by simp
      moreover
      from ss2 have "set (?map ss2) ⊆ A" by auto
      ultimately
      have "?sum ss2 <=_r ?s1" using x by - (rule pp_lub)
    }
```

```
    moreover
    from x ss2 have
      "?s2 =
      (if ∀(pc', s')∈set ss2. pc' ≠ pc + 1 ⟶ s' <=_r c!pc'
      then map snd [(p', t') ← ss2 . p' = pc + 1] ++_f x
      else ⊤)"
      by (rule merge_def)
    ultimately have ?thesis by simp
  }
  ultimately show ?thesis by (cases "?s1 = ⊤") auto
qed


lemma (in lbvc) wti_mono:
  assumes less: "s2 <=_r s1"
  assumes pc:   "pc < length φ"
  assumes s1:   "s1 ∈ A"
  assumes s2:   "s2 ∈ A"
  shows "wti c pc s2 <=_r wti c pc s1" (is "?s2' <=_r ?s1'")
proof -
  from mono pc s2 less have "step pc s2 <=|r| step pc s1" by (rule monoD)
  moreover
  from cert B_A pc have "c!Suc pc ∈ A" by (rule cert_okD3)
  moreover
  from pres s1 pc
  have "snd`set (step pc s1) ⊆ A" by (rule pres_typeD2)
  moreover
  from pres s2 pc
  have "snd`set (step pc s2) ⊆ A" by (rule pres_typeD2)
  ultimately
  show ?thesis by (simp add: wti merge_mono)
qed

lemma (in lbvc) wtc_mono:
  assumes less: "s2 <=_r s1"
  assumes pc:   "pc < length φ"
  assumes s1:   "s1 ∈ A"
  assumes s2:   "s2 ∈ A"
  shows "wtc c pc s2 <=_r wtc c pc s1" (is "?s2' <=_r ?s1'")
proof (cases "c!pc = ⊥")
  case True
  moreover from less pc s1 s2 have "wti c pc s2 <=_r wti c pc s1" by (rule wti_mono)
  ultimately show ?thesis by (simp add: wtc)
next
  case False
  have "?s1' = ⊤ ⟹ ?thesis" by simp
  moreover {
    assume "?s1' ≠ ⊤"
    with False have c: "s1 <=_r c!pc" by (simp add: wtc split: split_if_asm)
    with less have "s2 <=_r c!pc" ..
    with False c have ?thesis by (simp add: wtc)
  }
  ultimately show ?thesis by (cases "?s1' = ⊤") auto
qed
```

```
lemma (in lbv) top_le_conv [simp]:
  "⊤ <=_r x = (x = ⊤)"
  by (insert semilat) (simp add: top top_le_conv)

lemma (in lbv) neq_top [simp, elim]:
  "⟦ x <=_r y; y ≠ ⊤ ⟧ ⟹ x ≠ ⊤"
  by (cases "x = T") auto


lemma (in lbvc) stable_wti:
  assumes stable:  "stable r step φ pc"
  assumes pc:      "pc < length φ"
  shows "wti c pc (φ!pc) ≠ ⊤"
proof -
  let ?step = "step pc (φ!pc)"
  from stable
  have less: "∀(q,s')∈set ?step. s' <=_r φ!q" by (simp add: stable_def)

  from cert B_A pc
  have cert_suc: "c!Suc pc ∈ A" by (rule cert_okD3)
  moreover
  from phi pc have "φ!pc ∈ A" by simp
  from pres this pc
  have stepA: "snd`set ?step ⊆ A" by (rule pres_typeD2)
  ultimately
  have "merge c pc ?step (c!Suc pc) =
    (if ∀(pc',s')∈set ?step. pc'≠pc+1 ⟶ s' <=_r c!pc'
    then map snd [(p',t') ← ?step.p'=pc+1] ++_f c!Suc pc
    else ⊤)" unfolding mrg_def by (rule lbv.merge_def [OF lbvc.axioms(1), OF lbvc_axioms])
  moreover {
    fix pc' s' assume s': "(pc', s') ∈ set ?step" and suc_pc: "pc' ≠ pc+1"
    with less have "s' <=_r φ!pc'" by auto
    also
    from bounded pc s' have "pc' < length φ" by (rule boundedD)
    with s' suc_pc pc have "c!pc' = φ!pc'" ..
    hence "φ!pc' = c!pc'" ..
    finally have "s' <=_r c!pc'" .
  } hence "∀(pc',s')∈set ?step. pc'≠pc+1 ⟶ s' <=_r c!pc'" by auto
  moreover
  from pc have "Suc pc = length φ ∨ Suc pc < length φ" by auto
  hence "map snd [(p',t') ← ?step.p'=pc+1] ++_f c!Suc pc ≠ ⊤"
          (is "?map ++_f _ ≠ _")
  proof (rule disjE)
    assume pc': "Suc pc = length φ"
    with cert have "c!Suc pc = ⊥" by (simp add: cert_okD2)
    moreover
    from pc' bounded pc
    have "∀(p',t')∈set ?step. p'≠pc+1" by clarify (drule boundedD, auto)
    hence "[(p',t') ← ?step.p'=pc+1] = []" by (blast intro: filter_False)
    hence "?map = []" by simp
    ultimately show ?thesis by (simp add: B_neq_T)
  next
```

```
    assume pc': "Suc pc < length φ"
    from pc' phi have "φ!Suc pc ∈ A" by simp
    moreover note cert_suc
    moreover from stepA
    have "set ?map ⊆ A" by auto
    moreover
    have "⋀s. s ∈ set ?map ⟹ ∃t. (Suc pc, t) ∈ set ?step" by auto
    with less have "∀s' ∈ set ?map. s' <=_r φ!Suc pc" by auto
    moreover
    from pc' have "c!Suc pc <=_r φ!Suc pc"
      by (cases "c!Suc pc = ⊥") (auto dest: cert_approx)
    ultimately
    have "?map ++_f c!Suc pc <=_r φ!Suc pc" by (rule pp_lub)
    moreover
    from pc' phi have "φ!Suc pc ≠ ⊤" by simp
    ultimately
    show ?thesis by auto
  qed
  ultimately
  have "merge c pc ?step (c!Suc pc) ≠ ⊤" by simp
  thus ?thesis by (simp add: wti)
qed

lemma (in lbvc) wti_less:
  assumes stable:  "stable r step φ pc"
  assumes suc_pc:   "Suc pc < length φ"
  shows "wti c pc (φ!pc) <=_r φ!Suc pc" (is "?wti <=_r _")
proof -
  let ?step = "step pc (φ!pc)"

  from stable
  have less: "∀(q,s')∈set ?step. s' <=_r φ!q" by (simp add: stable_def)

  from suc_pc have pc: "pc < length φ" by simp
  with cert B_A have cert_suc: "c!Suc pc ∈ A" by (rule cert_okD3)
  moreover
  from phi pc have "φ!pc ∈ A" by simp
  with pres pc have stepA: "snd`set ?step ⊆ A" by - (rule pres_typeD2)
  moreover
  from stable pc have "?wti ≠ ⊤" by (rule stable_wti)
  hence "merge c pc ?step (c!Suc pc) ≠ ⊤" by (simp add: wti)
  ultimately
  have "merge c pc ?step (c!Suc pc) =
    map snd [(p',t')← ?step.p'=pc+1] ++_f c!Suc pc" by (rule merge_not_top_s)
  hence "?wti = ..." (is "_ = (?map ++_f _)" is "_ = ?sum") by (simp add: wti)
  also {
    from suc_pc phi have "φ!Suc pc ∈ A" by simp
    moreover note cert_suc
    moreover from stepA have "set ?map ⊆ A" by auto
    moreover
    have "⋀s. s ∈ set ?map ⟹ ∃t. (Suc pc, t) ∈ set ?step" by auto
    with less have "∀s' ∈ set ?map. s' <=_r φ!Suc pc" by auto
    moreover
    from suc_pc have "c!Suc pc <=_r φ!Suc pc"
```

```
      by (cases "c!Suc pc = ⊥") (auto dest: cert_approx)
    ultimately
    have "?sum <=_r φ!Suc pc" by (rule pp_lub)
  }
  finally show ?thesis .
qed

lemma (in lbvc) stable_wtc:
  assumes stable:  "stable r step phi pc"
  assumes pc:      "pc < length φ"
  shows "wtc c pc (φ!pc) ≠ ⊤"
proof -
  from stable pc have wti: "wti c pc (φ!pc) ≠ ⊤" by (rule stable_wti)
  show ?thesis
  proof (cases "c!pc = ⊥")
    case True with wti show ?thesis by (simp add: wtc)
  next
    case False
    with pc have "c!pc = φ!pc" ..
    with False wti show ?thesis by (simp add: wtc)
  qed
qed

lemma (in lbvc) wtc_less:
  assumes stable: "stable r step φ pc"
  assumes suc_pc: "Suc pc < length φ"
  shows "wtc c pc (φ!pc) <=_r φ!Suc pc" (is "?wtc <=_r _")
proof (cases "c!pc = ⊥")
  case True
  moreover from stable suc_pc have "wti c pc (φ!pc) <=_r φ!Suc pc"
    by (rule wti_less)
  ultimately show ?thesis by (simp add: wtc)
next
  case False
  from suc_pc have pc: "pc < length φ" by simp
  with stable have "?wtc ≠ ⊤" by (rule stable_wtc)
  with False have "?wtc = wti c pc (c!pc)"
    by (unfold wtc) (simp split: split_if_asm)
  also from pc False have "c!pc = φ!pc" ..
  finally have "?wtc = wti c pc (φ!pc)" .
  also from stable suc_pc have "wti c pc (φ!pc) <=_r φ!Suc pc" by (rule wti_less)
  finally show ?thesis .
qed


lemma (in lbvc) wt_step_wtl_lemma:
  assumes wt_step: "wt_step r ⊤ step φ"
  shows "⋀pc s. pc+length ls = length φ ⟹ s <=_r φ!pc ⟹ s ∈ A ⟹ s≠⊤ ⟹
              wtl ls c pc s ≠ ⊤"
   (is "⋀pc s. _ ⟹ _ ⟹ _ ⟹ _ ⟹ ?wtl ls pc s ≠ _")
proof (induct ls)
  fix pc s assume "s≠⊤" thus "?wtl [] pc s ≠ ⊤" by simp
next
  fix pc s i ls
```

```
  assume "⋀pc s. pc+length ls=length φ ⟹ s <=_r φ!pc ⟹ s ∈ A ⟹ s≠⊤ ⟹
                 ?wtl ls pc s ≠ ⊤"
  moreover
  assume pc_l: "pc + length (i#ls) = length φ"
  hence suc_pc_l: "Suc pc + length ls = length φ" by simp
  ultimately
  have IH: "⋀s. s <=_r φ!Suc pc ⟹ s ∈ A ⟹ s ≠ ⊤ ⟹ ?wtl ls (Suc pc) s ≠ ⊤" .

  from pc_l obtain pc: "pc < length φ" by simp
  with wt_step have stable: "stable r step φ pc" by (simp add: wt_step_def)
  from this pc have wt_phi: "wtc c pc (φ!pc) ≠ ⊤" by (rule stable_wtc)
  assume s_phi: "s <=_r φ!pc"
  from phi pc have phi_pc: "φ!pc ∈ A" by simp
  assume s: "s ∈ A"
  with s_phi pc phi_pc have wt_s_phi: "wtc c pc s <=_r wtc c pc (φ!pc)" by (rule wtc_mono)
  with wt_phi have wt_s: "wtc c pc s ≠ ⊤" by simp
  moreover
  assume s': "s ≠ ⊤"
  ultimately
  have "ls = [] ⟹ ?wtl (i#ls) pc s ≠ ⊤" by simp
  moreover {
    assume "ls ≠ []"
    with pc_l have suc_pc: "Suc pc < length φ" by (auto simp add: neq_Nil_conv)
    with stable have "wtc c pc (phi!pc) <=_r φ!Suc pc" by (rule wtc_less)
    with wt_s_phi have "wtc c pc s <=_r φ!Suc pc" by (rule trans_r)
    moreover
    from cert suc_pc have "c!pc ∈ A" "c!(pc+1) ∈ A"
      by (auto simp add: cert_ok_def)
    from pres this s pc have "wtc c pc s ∈ A" by (rule wtc_pres)
    ultimately
    have "?wtl ls (Suc pc) (wtc c pc s) ≠ ⊤" using IH wt_s by blast
    with s' wt_s have "?wtl (i#ls) pc s ≠ ⊤" by simp
  }
  ultimately show "?wtl (i#ls) pc s ≠ ⊤" by (cases ls) blast+
qed


theorem (in lbvc) wtl_complete:
  assumes wt: "wt_step r ⊤ step φ"
    and s: "s <=_r φ!0" "s ∈ A" "s ≠ ⊤"
    and len: "length ins = length phi"
  shows "wtl ins c 0 s ≠ ⊤"
proof -
  from len have "0+length ins = length phi" by simp
  from wt this s show ?thesis by (rule wt_step_wtl_lemma)
qed

end
```

Abstract Bytecode Verifier
Semilattices

4.13 The Java Type System as Semilattice

```
theory JType
imports "../DFA/Semilattices" "../J/WellForm"
begin

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

lemma superI:
  "G ⊢ C ≺C1 D ⟹ super G C = D"
  by (unfold super_def) (auto dest: subcls1D)

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

definition sup :: "'c prog ⇒ ty ⇒ ty ⇒ ty err" where
  "sup G T1 T2 ==
  case T1 of PrimT P1 ⇒ (case T2 of PrimT P2 ⇒
                          (if P1 = P2 then OK (PrimT P1) else Err) | RefT R ⇒ Err)
           | RefT R1 ⇒ (case T2 of PrimT P ⇒ Err | RefT R2 ⇒
  (case R1 of NullT ⇒ (case R2 of NullT ⇒ OK NT | ClassT C ⇒ OK (Class C))
            | ClassT C ⇒ (case R2 of NullT ⇒ OK (Class C)
                           | ClassT D ⇒ OK (Class (exec_lub (subcls1 G) (super G) C D)))))"

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

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

abbreviation "types G == Collect (is_type G)"

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

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

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

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

lemma is_type_conv:
  "ws_prog G ⟹ is_type G T = is_ty G T"
proof
  assume "is_type G T" "ws_prog G"
  thus "is_ty G T"
    by (rule is_tyI)
```

```
next
  assume wf: "ws_prog G" and
         ty: "is_ty G T"

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

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

lemma wf_converse_subcls1_impl_acc_subtype:
  "wf ((subcls1 G)^-1) ⟹ acc (subtype G)"
apply (unfold Semilat.acc_def lesssub_def)
apply (drule_tac p = "((subcls1 G)^-1) - Id" in wf_subset)
 apply auto
apply (drule wf_trancl)
apply (simp add: wf_eq_minimal)
apply clarify
apply (unfold lesub_def subtype_def)
apply (rename_tac M T)
```

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

lemma closed_err_types:
  "⟦ ws_prog G; single_valued (subcls1 G); acyclic (subcls1 G) ⟧
  ⟹ closed (err (types G)) (lift2 (sup G))"
  apply (unfold closed_def plussub_def lift2_def sup_def)
  apply (auto split: err.split)
  apply (drule is_tyI, assumption)
  apply (auto simp add: is_ty_def is_type_conv simp del: is_type.simps
              split: ty.split ref_ty.split)
  apply (blast dest!: is_lub_exec_lub is_lubD is_ubD intro!: is_ubI superI)
  done
```

```
lemma sup_subtype_greater:
  "⟦ ws_prog G; single_valued (subcls1 G); acyclic (subcls1 G);
      is_type G t1; is_type G t2; sup G t1 t2 = OK s ⟧
  ⟹ subtype G t1 s ∧ subtype G t2 s"
proof -
  assume ws_prog:       "ws_prog G"
  assume single_valued: "single_valued (subcls1 G)"
  assume acyclic:       "acyclic (subcls1 G)"

  { fix c1 c2
    assume is_class: "is_class G c1" "is_class G c2"
    with ws_prog
    obtain
      "G ⊢ c1 ≼C Object"
      "G ⊢ c2 ≼C Object"
      by (blast intro: subcls_C_Object)
    with ws_prog single_valued
    obtain u where
      "is_lub ((subcls1 G)^* ) c1 c2 u"
      by (blast dest: single_valued_has_lubs)
    moreover
    note acyclic
    moreover
    have "∀x y. G ⊢ x ≺C1 y ⟶ super G x = y"
      by (blast intro: superI)
    ultimately
    have "G ⊢ c1 ≼C exec_lub (subcls1 G) (super G) c1 c2 ∧
          G ⊢ c2 ≼C exec_lub (subcls1 G) (super G) c1 c2"
      by (simp add: exec_lub_conv) (blast dest: is_lubD is_ubD)
  } note this [simp]

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

lemma sup_subtype_smallest:
  "⟦ ws_prog G; single_valued (subcls1 G); acyclic (subcls1 G);
      is_type G a; is_type G b; is_type G c;
      subtype G a c; subtype G b c; sup G a b = OK d ⟧
  ⟹ subtype G d c"
proof -
  assume ws_prog:       "ws_prog G"
  assume single_valued: "single_valued (subcls1 G)"
  assume acyclic:       "acyclic (subcls1 G)"

  { fix c1 c2 D
    assume is_class: "is_class G c1" "is_class G c2"
    assume le: "G ⊢ c1 ≼C D" "G ⊢ c2 ≼C D"
    from ws_prog is_class
```

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

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

  assume "is_type G a" "is_type G b" "is_type G c"
         "subtype G a c" "subtype G b c" "sup G a b = OK d"
  thus ?thesis
    by (auto simp add: subtype_def sup_def
             split: ty.split_asm ref_ty.split_asm split_if_asm)
qed

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

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

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

  from ws_prog single_valued acyclic
  have
```

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

  moreover

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

  ultimately

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

lemma single_valued_subcls1:
  "ws_prog G ⟹ single_valued (subcls1 G)"
  by (auto simp add: ws_prog_def unique_def single_valued_def
    intro: subcls1I elim!: subcls1.cases)

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

end
```

4.14 The JVM Type System as Semilattice

```
theory JVMType
imports JType
begin

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


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

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

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

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

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

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

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

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

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

definition sup_state_opt :: "['code prog,state_type option,state_type option] ⇒ bool"

                   ("_ |- _ <=' _"  [71,71] 70) where
  "sup_state_opt G == Opt.le (sup_state G)"


notation (xsymbols)
```

```
  sup_ty_opt  ("_ ⊢ _ <=o _" [71,71] 70) and
  sup_loc  ("_ ⊢ _ <=l _" [71,71] 70) and
  sup_state  ("_ ⊢ _ <=s _" [71,71] 70) and
  sup_state_opt  ("_ ⊢ _ <=' _" [71,71] 70)


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


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

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

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

lemma zip_map [rule_format]:
  "∀a. length a = length b ⟶
  zip (map f a) (map g b) = map (λ(x,y). (f x, g y)) (zip a b)"
  apply (induct b)
   apply simp
  apply clarsimp
  apply (case_tac aa)
  apply simp+
  done

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

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

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

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

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

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


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


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

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

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

theorem sup_loc_refl [simp]:
  "G ⊢ t <=l t"
  by (simp add: sup_loc_def)

theorem sup_state_refl [simp]:
  "G ⊢ s <=s s"
```

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

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


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

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

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

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

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

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

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

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

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


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

```
qed

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

    fix x xs assume IH: "?P xs"

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

theorem all_nth_sup_loc:
  "∀b. length a = length b ⟶ (∀ n. n < length a ⟶ (G ⊢ (a!n) <=o (b!n)))
  ⟶ (G ⊢ a <=l b)" (is "?P a")
proof (induct a)
  show "?P []" by simp

  fix l ls assume IH: "?P ls"

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

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

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

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

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

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

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

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

    fix l ls assume IH: "?P ls"

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

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

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

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

  thus ?thesis by blast
qed


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

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


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

theorem sup_state_append_snd:
  "length a = length b ⟹
  (G ⊢ (i,a@x) <=s (j,b@y)) = ((G ⊢ (i,a) <=s (j,b)) ∧ (G ⊢ (i,x) <=s (j,y)))"
  by (auto simp add: sup_state_def stk_convert lesub_def Product.le_def sup_loc_append)

theorem sup_state_append_fst:
  "length a = length b ⟹
  (G ⊢ (a@x,i) <=s (b@y,j)) = ((G ⊢ (a,i) <=s (b,j)) ∧ (G ⊢ (x,i) <=s (y,j)))"
  by (auto simp add: sup_state_def stk_convert lesub_def Product.le_def sup_loc_append)

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

```
theorem sup_state_Cons2:
  "(G ⊢ (xt, a) <=s (y#yt, b)) =
   (∃x xt'. xt=x#xt' ∧ (G ⊢ x ≼ y) ∧ (G ⊢ (xt',a) <=s (yt,b)))"
  by (auto simp add: sup_state_def stk_convert lesub_def Product.le_def sup_loc_Cons2)

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

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


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

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

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

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

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


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

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

  hence "∀ n. n < length a ⟶ (G ⊢ (a!n) <=o (c!n))"
  proof (intro strip)
    fix n
    assume n: "n < length a"
```

```
    with G
    have "G ⊢ (a!n) <=o (b!n)"
      by - (rule sup_loc_nth)
    also
    from n G
    have "G ⊢ ... <=o (c!n)"
      by - (rule sup_loc_nth, auto dest: sup_loc_length)
    finally
    show "G ⊢ (a!n) <=o (c!n)" .
  qed

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


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

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

end
```

4.15 Effect of Instructions on the State Type

```
theory Effect
imports JVMType "../JVM/JVMExceptions"
begin

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

Program counter of successor instructions:

```
primrec succs :: "instr ⇒ p_count ⇒ p_count list" where
  "succs (Load idx) pc         = [pc+1]"
| "succs (Store idx) pc        = [pc+1]"
| "succs (LitPush v) pc        = [pc+1]"
| "succs (Getfield F C) pc     = [pc+1]"
| "succs (Putfield F C) pc     = [pc+1]"
| "succs (New C) pc            = [pc+1]"
| "succs (Checkcast C) pc      = [pc+1]"
| "succs Pop pc                = [pc+1]"
| "succs Dup pc                = [pc+1]"
| "succs Dup_x1 pc             = [pc+1]"
| "succs Dup_x2 pc             = [pc+1]"
| "succs Swap pc               = [pc+1]"
| "succs IAdd pc               = [pc+1]"
| "succs (Ifcmpeq b) pc        = [pc+1, nat (int pc + b)]"
| "succs (Goto b) pc           = [nat (int pc + b)]"
| "succs Return pc             = [pc]"
| "succs (Invoke C mn fpTs) pc = [pc+1]"
| "succs Throw pc              = [pc]"
```

Effect of instruction on the state type:

```
fun eff' :: "instr × jvm_prog × state_type ⇒ state_type"
where
"eff' (Load idx,  G, (ST, LT))          = (ok_val (LT ! idx) # ST, LT)" |
"eff' (Store idx, G, (ts#ST, LT))       = (ST, LT[idx:= OK ts])" |
"eff' (LitPush v, G, (ST, LT))           = (the (typeof (λv. None) v) # ST, LT)" |
"eff' (Getfield F C, G, (oT#ST, LT))    = (snd (the (field (G,C) F)) # ST, LT)" |
"eff' (Putfield F C, G, (vT#oT#ST, LT)) = (ST,LT)" |
"eff' (New C, G, (ST,LT))               = (Class C # ST, LT)" |
"eff' (Checkcast C, G, (RefT rt#ST,LT)) = (Class C # ST,LT)" |
"eff' (Pop, G, (ts#ST,LT))              = (ST,LT)" |
"eff' (Dup, G, (ts#ST,LT))              = (ts#ts#ST,LT)" |
"eff' (Dup_x1, G, (ts1#ts2#ST,LT))      = (ts1#ts2#ts1#ST,LT)" |
"eff' (Dup_x2, G, (ts1#ts2#ts3#ST,LT))  = (ts1#ts2#ts3#ts1#ST,LT)" |
"eff' (Swap, G, (ts1#ts2#ST,LT))        = (ts2#ts1#ST,LT)" |
"eff' (IAdd, G, (PrimT Integer#PrimT Integer#ST,LT))
                                         = (PrimT Integer#ST,LT)" |
"eff' (Ifcmpeq b, G, (ts1#ts2#ST,LT))   = (ST,LT)" |
"eff' (Goto b, G, s)                    = s" |
  — Return has no successor instruction in the same method
"eff' (Return, G, s)                    = s" |
  — Throw always terminates abruptly
"eff' (Throw, G, s)                     = s" |
"eff' (Invoke C mn fpTs, G, (ST,LT))    = (let ST' = drop (length fpTs) ST
```

```
  in  (fst (snd (the (method (G,C) (mn,fpTs))))#(tl ST'),LT))"
```

primrec *match_any :: "jvm_prog ⇒ p_count ⇒ exception_table ⇒ cname list"* **where**

```
  "match_any G pc [] = []"
| "match_any G pc (e#es) = (let (start_pc, end_pc, handler_pc, catch_type) = e;
                                es' = match_any G pc es
                            in
                            if start_pc <= pc ∧ pc < end_pc then catch_type#es' else es')"
```

primrec *match :: "jvm_prog ⇒ xcpt ⇒ p_count ⇒ exception_table ⇒ cname list"* **where**

```
  "match G X pc [] = []"
| "match G X pc (e#es) =
  (if match_exception_entry G (Xcpt X) pc e then [Xcpt X] else match G X pc es)"
```

lemma *match_some_entry:*

```
  "match G X pc et = (if ∃e ∈ set et. match_exception_entry G (Xcpt X) pc e then [Xcpt
X] else [])"
```

by *(induct et) auto*

fun

```
  xcpt_names :: "instr × jvm_prog × p_count × exception_table ⇒ cname list"
```

where

```
  "xcpt_names (Getfield F C, G, pc, et) = match G NullPointer pc et"
| "xcpt_names (Putfield F C, G, pc, et) = match G NullPointer pc et"
| "xcpt_names (New C, G, pc, et)        = match G OutOfMemory pc et"
| "xcpt_names (Checkcast C, G, pc, et)  = match G ClassCast pc et"
| "xcpt_names (Throw, G, pc, et)        = match_any G pc et"
| "xcpt_names (Invoke C m p, G, pc, et) = match_any G pc et"
| "xcpt_names (i, G, pc, et)            = []"
```

definition *xcpt_eff :: "instr ⇒ jvm_prog ⇒ p_count ⇒ state_type option ⇒ exception_table ⇒ succ_type"* **where**

```
  "xcpt_eff i G pc s et ==
   map (λC. (the (match_exception_table G C pc et), case s of None ⇒ None | Some s' ⇒
Some ([Class C], snd s')))
       (xcpt_names (i,G,pc,et))"
```

definition *norm_eff :: "instr ⇒ jvm_prog ⇒ state_type option ⇒ state_type option"* **where**

```
  "norm_eff i G == Option.map (λs. eff' (i,G,s))"
```

definition *eff :: "instr ⇒ jvm_prog ⇒ p_count ⇒ exception_table ⇒ state_type option ⇒ succ_type"* **where**

```
  "eff i G pc et s == (map (λpc'. (pc',norm_eff i G s)) (succs i pc)) @ (xcpt_eff i G
pc s et)"
```

definition *isPrimT :: "ty ⇒ bool"* **where**

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

definition *isRefT :: "ty ⇒ bool"* **where**

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

```
lemma isPrimT [simp]:
  "isPrimT T = (∃T'. T = PrimT T')" by (simp add: isPrimT_def split: ty.splits)

lemma isRefT [simp]:
  "isRefT T = (∃T'. T = RefT T')" by (simp add: isRefT_def split: ty.splits)


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

Conditions under which eff is applicable:

```
fun
app' :: "instr × jvm_prog × p_count × nat × ty × state_type ⇒ bool"
where
"app' (Load idx, G, pc, maxs, rT, s) =
  (idx < length (snd s) ∧ (snd s) ! idx ≠ Err ∧ length (fst s) < maxs)" |
"app' (Store idx, G, pc, maxs, rT, (ts#ST, LT)) =
  (idx < length LT)" |
"app' (LitPush v, G, pc, maxs, rT, s) =
  (length (fst s) < maxs ∧ typeof (λt. None) v ≠ None)" |
"app' (Getfield F C, G, pc, maxs, rT, (oT#ST, LT)) =
  (is_class G C ∧ field (G,C) F ≠ None ∧ fst (the (field (G,C) F)) = C ∧
  G ⊢ oT ≼ (Class C))" |
"app' (Putfield F C, G, pc, maxs, rT, (vT#oT#ST, LT)) =
  (is_class G C ∧ field (G,C) F ≠ None ∧ fst (the (field (G,C) F)) = C ∧
  G ⊢ oT ≼ (Class C) ∧ G ⊢ vT ≼ (snd (the (field (G,C) F))))" |
"app' (New C, G, pc, maxs, rT, s) =
  (is_class G C ∧ length (fst s) < maxs)" |
"app' (Checkcast C, G, pc, maxs, rT, (RefT rt#ST,LT)) =
  (is_class G C)" |
"app' (Pop, G, pc, maxs, rT, (ts#ST,LT)) =
  True" |
"app' (Dup, G, pc, maxs, rT, (ts#ST,LT)) =
  (1+length ST < maxs)" |
"app' (Dup_x1, G, pc, maxs, rT, (ts1#ts2#ST,LT)) =
  (2+length ST < maxs)" |
"app' (Dup_x2, G, pc, maxs, rT, (ts1#ts2#ts3#ST,LT)) =
  (3+length ST < maxs)" |
"app' (Swap, G, pc, maxs, rT, (ts1#ts2#ST,LT)) =
  True" |
"app' (IAdd, G, pc, maxs, rT, (PrimT Integer#PrimT Integer#ST,LT)) =
  True" |
"app' (Ifcmpeq b, G, pc, maxs, rT, (ts#ts'#ST,LT)) =
  (0 ≤ int pc + b ∧ (isPrimT ts ∧ ts' = ts ∨ isRefT ts ∧ isRefT ts'))" |
"app' (Goto b, G, pc, maxs, rT, s) =
  (0 ≤ int pc + b)" |
"app' (Return, G, pc, maxs, rT, (T#ST,LT)) =
  (G ⊢ T ≼ rT)" |
"app' (Throw, G, pc, maxs, rT, (T#ST,LT)) =
  isRefT T" |
"app' (Invoke C mn fpTs, G, pc, maxs, rT, s) =
  (length fpTs < length (fst s) ∧
  (let apTs = rev (take (length fpTs) (fst s));
       X    = hd (drop (length fpTs) (fst s))
```

```
  in
     G ⊢ X ≼ Class C ∧ is_class G C ∧ method (G,C) (mn,fpTs) ≠ None ∧
     list_all2 (λx y. G ⊢ x ≼ y) apTs fpTs))" |

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

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

definition app :: "instr ⇒ jvm_prog ⇒ nat ⇒ ty ⇒ nat ⇒ exception_table ⇒ state_type
option ⇒ bool" where
  "app i G maxs rT pc et s == case s of None ⇒ True | Some t ⇒ app' (i,G,pc,maxs,rT,t)
∧ xcpt_app i G pc et"


lemma match_any_match_table:
  "C ∈ set (match_any G pc et) ⟹ match_exception_table G C pc et ≠ None"
  apply (induct et)
   apply simp
  apply simp
  apply clarify
  apply (simp split: split_if_asm)
  apply (auto simp add: match_exception_entry_def)
  done

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

lemma xcpt_names_in_et:
  "C ∈ set (xcpt_names (i,G,pc,et)) ⟹
  ∃e ∈ set et. the (match_exception_table G C pc et) = fst (snd (snd e))"
  apply (cases i)
  apply (auto dest!: match_any_match_table match_X_match_table
              dest: match_exception_table_in_et)
  done


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

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

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

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

lemmas [simp] = app_def xcpt_app_def
```

simp rules for app

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


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

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

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

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

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

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

lemma appCheckcast[simp]:
  "(app (Checkcast C) G maxs rT pc et (Some s)) =
  (∃rT ST LT. s = (RefT rT#ST,LT) ∧ is_class G C ∧
```

```
  (∀x ∈ set (match G ClassCast pc et). is_class G x))"
  by (cases s, cases "fst s", simp) (cases "hd (fst s)", auto)

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


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

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


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


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


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


lemma appIAdd[simp]:
"app IAdd G maxs rT pc et (Some s) = (∃ ST LT. s = (PrimT Integer#PrimT Integer#ST,LT))"
  (is "?app s = ?P s")
proof (cases s)
  case (Pair a b)
  have "?app (a,b) = ?P (a,b)"
  proof (cases a)
    fix t ts assume a: "a = t#ts"
    show ?thesis
    proof (cases t)
      fix p assume p: "t = PrimT p"
      show ?thesis
      proof (cases p)
        assume ip: "p = Integer"
        show ?thesis
        proof (cases ts)
          fix t' ts' assume t': "ts = t' # ts'"
          show ?thesis
          proof (cases t')
            fix p' assume "t' = PrimT p'"
            with t' ip p a
            show ?thesis by (cases p') auto
          qed (auto simp add: a p ip t')
```

```
      qed (auto simp add: a p ip)
    qed (auto simp add: a p)
  qed (auto simp add: a)
 qed auto
 with Pair show ?thesis by simp
qed


lemma appIfcmpeq[simp]:
"app (Ifcmpeq b) G maxs rT pc et (Some s) =
  (∃ts1 ts2 ST LT. s = (ts1#ts2#ST,LT) ∧ 0 ≤ int pc + b ∧
  ((∃ p. ts1 = PrimT p ∧ ts2 = PrimT p) ∨ (∃r r'. ts1 = RefT r ∧ ts2 = RefT r')))"
  by (cases s, cases "2 <length (fst s)", auto dest!: 1 2)

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

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

lemma appThrow[simp]:
  "app Throw G maxs rT pc et (Some s) =
  (∃T ST LT r. s=(T#ST,LT) ∧ T = RefT r ∧ (∀C ∈ set (match_any G pc et). is_class G
C))"
  by (cases s, cases "2 < length (fst s)", auto dest: 1 2)

lemma appInvoke[simp]:
"app (Invoke C mn fpTs) G maxs rT pc et (Some s) = (∃apTs X ST LT mD' rT' b'.
  s = ((rev apTs) @ (X # ST), LT) ∧ length apTs = length fpTs ∧ is_class G C ∧
  G ⊢ X ≼ Class C ∧ (∀(aT,fT)∈set(zip apTs fpTs). G ⊢ aT ≼ fT) ∧
  method (G,C) (mn,fpTs) = Some (mD', rT', b') ∧
  (∀C ∈ set (match_any G pc et). is_class G C))" (is "?app s = ?P s")
proof (cases s)
  note list_all2_def [simp]
  case (Pair a b)
  have "?app (a,b) ⟹ ?P (a,b)"
  proof -
    assume app: "?app (a,b)"
    hence "a = (rev (rev (take (length fpTs) a))) @ (drop (length fpTs) a) ∧
           length fpTs < length a" (is "?a ∧ ?l")
      by auto
    hence "?a ∧ 0 < length (drop (length fpTs) a)" (is "?a ∧ ?l")
      by auto
    hence "?a ∧ ?l ∧ length (rev (take (length fpTs) a)) = length fpTs"
      by (auto)
    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 clarsimp blast
  qed
  with Pair
  have "?app s ⟹ ?P s" by (simp only:)
  moreover
  have "?P s ⟹ ?app s" by (clarsimp simp add: min_max.inf_absorb2)
  ultimately
  show ?thesis by (rule iffI)
qed

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


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

lemmas [simp del] = app_def xcpt_app_def

end
```

4.16 Monotonicity of eff and app

```
theory EffectMono
imports Effect
begin

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


lemma sup_loc_some [rule_format]:
"∀y n. (G ⊢ b <=l y) ⟶ n < length y ⟶ y!n = OK t ⟶
  (∃t. b!n = OK t ∧ (G ⊢ (b!n) <=o (y!n)))"
proof (induct b)
  case Nil
  show ?case by simp
next
  case (Cons a list)
  show ?case
  proof (clarsimp simp add: list_all2_Cons1 sup_loc_def Listn.le_def lesub_def)
    fix z zs n
    assume *:
      "G ⊢ a <=o z" "list_all2 (sup_ty_opt G) list zs"
      "n < Suc (length list)" "(z # zs) ! n = OK t"

    show "(∃t. (a # list) ! n = OK t) ∧ G ⊢(a # list) ! n <=o OK t"
    proof (cases n)
      case 0
      with * show ?thesis by (simp add: sup_ty_opt_OK)
    next
      case Suc
      with Cons *
      show ?thesis by (simp add: sup_loc_def Listn.le_def lesub_def)
    qed
  qed
qed


lemma all_widen_is_sup_loc:
"∀b. length a = length b ⟶
     (∀(x, y)∈set (zip a b). G ⊢ x ≼ y) = (G ⊢ (map OK a) <=l (map OK b))"
 (is "∀b. length a = length b ⟶ ?Q a b" is "?P a")
proof (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)"
proof (induct x)
  case Nil
  show ?case by simp
next
  case (Cons l ls)

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

    show "∃a b. l # ls = a @ b ∧ length a = n"
    proof (cases n)
      assume "n=0" thus ?thesis by simp
    next
      fix n' assume s: "n = Suc n'"
      with l have  "n' ≤ length ls" by simp
      hence "∃a b. ls = a @ b ∧ length a = n'" by (rule Cons [rule_format])
      then obtain a b where "ls = a @ b" "length a = n'" by iprover
      with s have "l # ls = (l#a) @ b ∧ length (l#a) = n" by simp
      thus ?thesis by blast
    qed
  qed
qed

lemma rev_append_cons:
"n < length x ⟹ ∃a b c. x = (rev a) @ b # c ∧ length a = n"
proof -
  assume n: "n < length x"
  hence "n ≤ length x" by simp
  hence "∃a b. x = a @ b ∧ length a = n" by (rule append_length_n)
  then obtain r d where x: "x = r@d" "length r = n" by iprover
  with n have "∃b c. d = b#c" by (simp add: neq_Nil_conv)
  then obtain b c where "d = b#c" by iprover
  with x have "x = (rev (rev r)) @ b # c ∧ length (rev r) = n" by simp
  thus ?thesis by blast
qed

lemma sup_loc_length_map:
  "G ⊢ map f a <=l map g b ⟹ length a = length b"
proof -
  assume "G ⊢ map f a <=l map g b"
  hence "length (map f a) = length (map g b)" by (rule sup_loc_length)
  thus ?thesis by simp
qed

lemmas [iff] = not_Err_eq

lemma app_mono:
```

```
"⟦G ⊢ s <=' s'; app i G m rT pc et s'⟧ ⟹ app i G m rT pc et s"
proof -

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

    note [simp] = sup_loc_length sup_loc_length_map

    have "app i G m rT pc et (Some s2)"
    proof (cases i)
      case Load

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

      with G Load app show ?thesis
        by (cases s2) (auto simp add: sup_state_conv dest: sup_loc_some)
    next
      case Store
      with G app show ?thesis
        by (cases s2) (auto simp add: sup_loc_Cons2 sup_state_conv)
    next
      case LitPush
      with G app show ?thesis by (cases s2) (auto simp add: sup_state_conv)
    next
      case New
      with G app show ?thesis by (cases s2) (auto simp add: sup_state_conv)
    next
      case Getfield
      with app G show ?thesis
        by (cases s2) (clarsimp simp add: sup_state_Cons2, rule widen_trans)
    next
      case (Putfield vname cname)

      with app
      obtain vT oT ST LT b
        where s1: "s1 = (vT # oT # ST, LT)" and
                  "field (G, cname) vname = Some (cname, b)"
                  "is_class G cname" and
              oT: "G⊢ oT≼ (Class cname)" and
              vT: "G⊢ vT≼ b" and
              xc: "Ball (set (match G NullPointer pc et)) (is_class G)"
        by force
      moreover
      from s1 G
      obtain vT' oT' ST' LT'
        where s2:  "s2 = (vT' # oT' # ST', LT')" and
              oT': "G⊢ oT' ≼ oT" and
              vT': "G⊢ vT' ≼ vT"
        by - (cases s2, simp add: sup_state_Cons2, elim exE conjE, simp, rule that)
      moreover
      from vT' vT
      have "G ⊢ vT' ≼ b" by (rule widen_trans)
```

```
    moreover
    from oT' oT
    have "G⊢ oT' ≼ (Class cname)" by (rule widen_trans)
    ultimately
    show ?thesis by (auto simp add: Putfield xc)
  next
    case Checkcast
    with app G show ?thesis
      by (cases s2) (auto intro!: widen_RefT2 simp add: sup_state_Cons2)
  next
    case Return
    with app G show ?thesis
      by (cases s2) (auto simp add: sup_state_Cons2, rule widen_trans)
  next
    case Pop
    with app G show ?thesis
      by (cases s2) (clarsimp simp add: sup_state_Cons2)
  next
    case Dup
    with app G show ?thesis
      by (cases s2) (clarsimp simp add: sup_state_Cons2,
          auto dest: sup_state_length)
  next
    case Dup_x1
    with app G show ?thesis
      by (cases s2) (clarsimp simp add: sup_state_Cons2,
          auto dest: sup_state_length)
  next
    case Dup_x2
    with app G show ?thesis
      by (cases s2) (clarsimp simp add: sup_state_Cons2,
          auto dest: sup_state_length)
  next
    case Swap
    with app G show ?thesis
      by (cases s2) (auto 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 cname mname list)

    with app
    obtain apTs X ST LT mD' rT' b' where
      s1: "s1 = (rev apTs @ X # ST, LT)" and
      l:  "length apTs = length list" and
```

```
    c:  "is_class G cname" and
    C:  "G ⊢ X ≼ Class cname" and
    w:  "∀(x, y) ∈ set (zip apTs list). G ⊢ x ≼ y" and
    m:  "method (G, cname) (mname, list) = Some (mD', rT', b')" and
    x:  "∀C ∈ set (match_any G pc et). is_class G C"
    by (simp del: not_None_eq, elim exE conjE) (rule that)

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

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

  from this s1 s2 G
  obtain
    G': "G ⊢ (apTs',LT') <=s (apTs,LT)" and
    X : "G ⊢  X' ≼ X" and "G ⊢ (ST',LT') <=s (ST,LT)"
    by (simp add: sup_state_rev_fst sup_state_append_fst sup_state_Cons1)

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

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

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

  from Invoke s2 l' w' C' m c x
  show ?thesis
    by (simp del: split_paired_Ex) blast
next
  case Throw
  with app G show ?thesis
    by (cases s2, clarsimp simp add: sup_state_Cons2 widen_RefT2)
```

```
    qed
  } note this [simp]

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

lemmas [simp del] = split_paired_Ex

lemma eff'_mono:
"⟦ app i G m rT pc et (Some s2); G ⊢ s1 <=s s2 ⟧ ⟹
  G ⊢ eff' (i,G,s1) <=s eff' (i,G,s2)"
proof (cases s1, cases s2)
  fix a1 b1 a2 b2
  assume s: "s1 = (a1,b1)" "s2 = (a2,b2)"
  assume app2: "app i G m rT pc et (Some s2)"
  assume G: "G ⊢ s1 <=s s2"

  note [simp] = eff_def

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

  show ?thesis
  proof (cases i)
    case (Load n)

    with s app1
    obtain y where
       y:  "n < length b1" "b1 ! n = OK y" by clarsimp

    from Load s app2
    obtain y' where
       y': "n < length b2" "b2 ! n = OK y'" by clarsimp

    from G s
    have "G ⊢ b1 <=l b2" by (simp add: sup_state_conv)

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

    with Load G y y' s app1 app2
    show ?thesis by (clarsimp simp add: sup_state_conv)
  next
    case Store
    with G s app1 app2
    show ?thesis
      by (clarsimp simp add: sup_state_conv sup_loc_update)
  next
    case LitPush
    with G s app1 app2
    show ?thesis
```

```
    by (clarsimp simp add: sup_state_Cons1)
next
  case New
  with G s app1 app2
  show ?thesis
    by (clarsimp simp add: sup_state_Cons1)
next
  case Getfield
  with G s app1 app2
  show ?thesis
    by (clarsimp simp add: sup_state_Cons1)
next
  case Putfield
  with G s app1 app2
  show ?thesis
    by (clarsimp simp add: sup_state_Cons1)
next
  case Checkcast
  with G s app1 app2
  show ?thesis
    by (clarsimp simp add: sup_state_Cons1)
next
  case (Invoke cname mname list)

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

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

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

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

  moreover

  obtain b1' b2' where eff':
    "b1' = snd (eff' (i,G,s1))"
    "b2' = snd (eff' (i,G,s2))" by simp

  from Invoke G s eff' app1 app2
  obtain "b1 = b1'" "b2 = b2'" by simp

  ultimately
```

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

    with Invoke G s app1 app2 eff' s1 s2 l l'
    show ?thesis
      by (clarsimp simp add: sup_state_conv)
  next
    case Return
    with G
    show ?thesis
      by simp
  next
    case Pop
    with G s app1 app2
    show ?thesis
      by (clarsimp simp add: sup_state_Cons1)
  next
    case Dup
    with G s app1 app2
    show ?thesis
      by (clarsimp simp add: sup_state_Cons1)
  next
    case Dup_x1
    with G s app1 app2
    show ?thesis
      by (clarsimp simp add: sup_state_Cons1)
  next
    case Dup_x2
    with G s app1 app2
    show ?thesis
      by (clarsimp simp add: sup_state_Cons1)
  next
    case Swap
    with G s app1 app2
    show ?thesis
      by (clarsimp simp add: sup_state_Cons1)
  next
    case IAdd
    with G s app1 app2
    show ?thesis
      by (clarsimp simp add: sup_state_Cons1)
  next
    case Goto
    with G s app1 app2
    show ?thesis by simp
  next
    case Ifcmpeq
    with G s app1 app2
    show ?thesis
      by (clarsimp simp add: sup_state_Cons1)
  next
    case Throw
    with G
    show ?thesis
      by simp
```

```
  qed
qed

lemmas [iff del] = not_Err_eq

end
```

4.17 The Bytecode Verifier

theory *BVSpec*
imports *Effect*
begin

This theory contains a specification of the BV. The specification describes correct typings of method bodies; it corresponds to type *checking*.

constdefs

— The program counter will always be inside the method:

```
check_bounded :: "instr list ⇒ exception_table ⇒ bool"
"check_bounded ins et ≡
(∀pc < length ins. ∀pc' ∈ set (succs (ins!pc) pc). pc' < length ins) ∧
                   (∀e ∈ set et. fst (snd (snd e)) < length ins)"
```

— The method type only contains declared classes:

```
check_types :: "jvm_prog ⇒ nat ⇒ nat ⇒ JVMType.state list ⇒ bool"
"check_types G mxs mxr phi ≡ set phi ⊆ states G mxs mxr"
```

— An instruction is welltyped if it is applicable and its effect
— is compatible with the type at all successor instructions:

```
wt_instr :: "[instr,jvm_prog,ty,method_type,nat,p_count,
              exception_table,p_count] ⇒ bool"
"wt_instr i G rT phi mxs max_pc et pc ≡
app i G mxs rT pc et (phi!pc) ∧
(∀(pc',s') ∈ set (eff i G pc et (phi!pc)). pc' < max_pc ∧ G ⊢ s' <=' phi!pc')"
```

— The type at *pc=0* conforms to the method calling convention:

```
wt_start :: "[jvm_prog,cname,ty list,nat,method_type] ⇒ bool"
"wt_start G C pTs mxl phi ==
G ⊢ Some ([],(OK (Class C))#((map OK pTs))@(replicate mxl Err)) <=' phi!0"
```

— A method is welltyped if the body is not empty, if execution does not
— leave the body, if the method type covers all instructions and mentions
— declared classes only, if the method calling convention is respected, and
— if all instructions are welltyped.

```
wt_method :: "[jvm_prog,cname,ty list,ty,nat,nat,instr list,
               exception_table,method_type] ⇒ bool"
"wt_method G C pTs rT mxs mxl ins et phi ≡
let max_pc = length ins in
0 < max_pc ∧
length phi = length ins ∧
check_bounded ins et ∧
check_types G mxs (1+length pTs+mxl) (map OK phi) ∧
wt_start G C pTs mxl phi ∧
(∀pc. pc<max_pc ⟶ wt_instr (ins!pc) G rT phi mxs max_pc et pc)"
```

— A program is welltyped if it is wellformed and all methods are welltyped

```
wt_jvm_prog :: "[jvm_prog,prog_type] ⇒ bool"
"wt_jvm_prog G phi ==
wf_prog (λG C (sig,rT,(maxs,maxl,b,et)).
         wt_method G C (snd sig) rT maxs maxl b et (phi C sig)) G"
```

```
lemma check_boundedD:
  "⟦ check_bounded ins et; pc < length ins;
    (pc',s') ∈ set (eff (ins!pc) G pc et s)  ⟧ ⟹
  pc' < length ins"
  apply (unfold eff_def)
  apply simp
  apply (unfold check_bounded_def)
  apply clarify
  apply (erule disjE)
   apply blast
  apply (erule allE, erule impE, assumption)
  apply (unfold xcpt_eff_def)
  apply clarsimp
  apply (drule xcpt_names_in_et)
  apply clarify
  apply (drule bspec, assumption)
  apply simp
  done

lemma wt_jvm_progD:
  "wt_jvm_prog G phi ⟹ (∃wt. wf_prog wt G)"
  by (unfold wt_jvm_prog_def, blast)

lemma wt_jvm_prog_impl_wt_instr:
  "⟦ wt_jvm_prog G phi; is_class G C;
      method (G,C) sig = Some (C,rT,maxs,maxl,ins,et); pc < length ins ⟧
  ⟹ wt_instr (ins!pc) G rT (phi C sig) maxs (length ins) et pc"
  by (unfold wt_jvm_prog_def, drule method_wf_mdecl,
      simp, simp, simp add: wf_mdecl_def wt_method_def)
```

We could leave out the check `pc' < max_pc` in the definition of `wt_instr` in the context of `wt_method`.

```
lemma wt_instr_def2:
  "⟦ wt_jvm_prog G Phi; is_class G C;
      method (G,C) sig = Some (C,rT,maxs,maxl,ins,et); pc < length ins;
      i = ins!pc; phi = Phi C sig; max_pc = length ins ⟧
  ⟹ wt_instr i G rT phi maxs max_pc et pc =
     (app i G maxs rT pc et (phi!pc) ∧
     (∀(pc',s') ∈ set (eff i G pc et (phi!pc)). G ⊢ s' <=' phi!pc'))"
apply (simp add: wt_instr_def)
apply (unfold wt_jvm_prog_def)
apply (drule method_wf_mdecl)
apply (simp, simp, simp add: wf_mdecl_def wt_method_def)
apply (auto dest: check_boundedD)
done

lemma wt_jvm_prog_impl_wt_start:
  "⟦ wt_jvm_prog G phi; is_class G C;
      method (G,C) sig = Some (C,rT,maxs,maxl,ins,et) ⟧ ⟹
  0 < (length ins) ∧ wt_start G C (snd sig) maxl (phi C sig)"
  by (unfold wt_jvm_prog_def, drule method_wf_mdecl,
      simp, simp, simp add: wf_mdecl_def wt_method_def)

end
```

4.18 The Typing Framework for the JVM

```
theory Typing_Framework_JVM
imports "../DFA/Abstract_BV" JVMType EffectMono BVSpec
begin

definition exec :: "jvm_prog ⇒ nat ⇒ ty ⇒ exception_table ⇒ instr list ⇒ JVMType.state
step_type" where
  "exec G maxs rT et bs ==
  err_step (size bs) (λpc. app (bs!pc) G maxs rT pc et) (λpc. eff (bs!pc) G pc et)"

definition opt_states :: "'c prog ⇒ nat ⇒ nat ⇒ (ty list × ty err list) option set"
where
  "opt_states G maxs maxr ≡ opt (⋃{list n (types G) |n. n ≤ maxs} × list maxr (err
(types G)))"
```

4.18.1 Executability of *check_bounded*

```
consts
  list_all'_rec :: "('a ⇒ nat ⇒ bool) ⇒ nat ⇒ 'a list ⇒ bool"
primrec
  "list_all'_rec P n []     = True"
  "list_all'_rec P n (x#xs) = (P x n ∧ list_all'_rec P (Suc n) xs)"

definition list_all' :: "('a ⇒ nat ⇒ bool) ⇒ 'a list ⇒ bool" where
  "list_all' P xs ≡ list_all'_rec P 0 xs"

lemma list_all'_rec:
  "⋀n. list_all'_rec P n xs = (∀p < size xs. P (xs!p) (p+n))"
  apply (induct xs)
  apply auto
  apply (case_tac p)
  apply auto
  done

lemma list_all' [iff]:
  "list_all' P xs = (∀n < size xs. P (xs!n) n)"
  by (unfold list_all'_def) (simp add: list_all'_rec)

lemma [code]:
  "check_bounded ins et =
  (list_all' (λi pc. list_all (λpc'. pc' < length ins) (succs i pc)) ins ∧
   list_all (λe. fst (snd (snd e)) < length ins) et)"
  by (simp add: list_all_iff check_bounded_def)
```

4.18.2 Connecting JVM and Framework

```
lemma check_bounded_is_bounded:
  "check_bounded ins et ⟹ bounded (λpc. eff (ins!pc) G pc et) (length ins)"
  by (unfold bounded_def) (blast dest: check_boundedD)

lemma special_ex_swap_lemma [iff]:
  "(? X. (? n. X = A n & P n) & Q X) = (? n. Q(A n) & P n)"
  by blast
```

```
lemmas [iff del] = not_None_eq

theorem exec_pres_type:
  "wf_prog wf_mb S ⟹
  pres_type (exec S maxs rT et bs) (size bs) (states S maxs maxr)"
  apply (unfold exec_def JVM_states_unfold)
  apply (rule pres_type_lift)
  apply clarify
  apply (case_tac s)
   apply simp
   apply (drule effNone)
   apply simp
  apply (simp add: eff_def xcpt_eff_def norm_eff_def)
  apply (case_tac "bs!p")

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

  apply (fastsimp simp add: not_None_eq)

  apply (fastsimp simp add: not_None_eq typeof_empty_is_type)

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

  apply clarsimp
  apply (erule disjE)
   apply (fastsimp dest: field_fields fields_is_type)
  apply (simp add: match_some_entry split: split_if_asm)
  apply (rule_tac x=1 in exI)
  apply fastsimp

  apply clarsimp
  apply (erule disjE)
   apply fastsimp
  apply (simp add: match_some_entry split: split_if_asm)
  apply (rule_tac x=1 in exI)
  apply fastsimp

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

  defer

  apply fastsimp
```

```
  apply fastsimp

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

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

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

  apply fastsimp
  apply fastsimp
  apply fastsimp
  apply fastsimp

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

  apply (erule disjE)
   apply clarsimp
   apply (drule method_wf_mdecl, assumption+)
   apply (clarsimp simp add: wf_mdecl_def wf_mhead_def)
   apply fastsimp
  apply clarsimp
  apply (rule_tac x=1 in exI)
  apply fastsimp
  done

lemmas [iff] = not_None_eq

lemma sup_state_opt_unfold:
  "sup_state_opt G ≡ Opt.le (Product.le (Listn.le (subtype G)) (Listn.le (Err.le (subtype
G))))"
  by (simp add: sup_state_opt_def sup_state_def sup_loc_def sup_ty_opt_def)


lemma app_mono:
  "app_mono (sup_state_opt G) (λpc. app (bs!pc) G maxs rT pc et) (length bs) (opt_states
G maxs maxr)"
  by (unfold app_mono_def lesub_def) (blast intro: EffectMono.app_mono)


lemma list_appendI:
  "⟦a ∈ list x A; b ∈ list y A⟧ ⟹ a @ b ∈ list (x+y) A"
  apply (unfold list_def)
  apply (simp (no_asm))
```

```
  apply blast
  done

lemma list_map [simp]:
  "(map f xs ∈ list (length xs) A) = (f ` set xs ⊆ A)"
  apply (unfold list_def)
  apply simp
  done

lemma [iff]:
  "(OK ` A ⊆ err B) = (A ⊆ B)"
  apply (unfold err_def)
  apply blast
  done

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

lemma lesubstep_type_simple:
  "a <=[Product.le (op =) r] b ⟹ a <=|r| b"
  apply (unfold lesubstep_type_def)
  apply clarify
  apply (simp add: set_conv_nth)
  apply clarify
  apply (drule le_listD, assumption)
  apply (clarsimp simp add: lesub_def Product.le_def)
  apply (rule exI)
  apply (rule conjI)
   apply (rule exI)
   apply (rule conjI)
    apply (rule sym)
    apply assumption
   apply assumption
  apply assumption
  done


lemma eff_mono:
  "⟦p < length bs; s <=_(sup_state_opt G) t; app (bs!p) G maxs rT pc et t⟧
  ⟹ eff (bs!p) G p et s <=|sup_state_opt G| eff (bs!p) G p et t"
  apply (unfold eff_def)
  apply (rule lesubstep_type_simple)
  apply (rule le_list_appendI)
   apply (simp add: norm_eff_def)
   apply (rule le_listI)
    apply simp
   apply simp
   apply (simp add: lesub_def)
   apply (case_tac s)
    apply simp
   apply (simp del: split_paired_All split_paired_Ex)
   apply (elim exE conjE)
   apply simp
```

```
   apply (drule eff'_mono, assumption)
   apply assumption
  apply (simp add: xcpt_eff_def)
  apply (rule le_listI)
    apply simp
  apply simp
  apply (simp add: lesub_def)
  apply (case_tac s)
   apply simp
  apply simp
  apply (case_tac t)
   apply simp
  apply (clarsimp simp add: sup_state_conv)
  done

lemma order_sup_state_opt:
  "ws_prog G ⟹ order (sup_state_opt G)"
  by (unfold sup_state_opt_unfold) (blast dest: acyclic_subcls1 order_widen)

theorem exec_mono:
  "ws_prog G ⟹ bounded (exec G maxs rT et bs) (size bs) ⟹
  mono (JVMType.le G maxs maxr) (exec G maxs rT et bs) (size bs) (states G maxs maxr)"

  apply (unfold exec_def JVM_le_unfold JVM_states_unfold)
  apply (rule mono_lift)
     apply (fold sup_state_opt_unfold opt_states_def)
     apply (erule order_sup_state_opt)
    apply (rule app_mono)
   apply assumption
  apply clarify
  apply (rule eff_mono)
  apply assumption+
  done

theorem semilat_JVM_slI:
  "ws_prog G ⟹ semilat (JVMType.sl G maxs maxr)"
  apply (unfold JVMType.sl_def stk_esl_def reg_sl_def)
  apply (rule semilat_opt)
  apply (rule err_semilat_Product_esl)
  apply (rule err_semilat_upto_esl)
  apply (rule err_semilat_JType_esl, assumption+)
  apply (rule err_semilat_eslI)
  apply (rule Listn_sl)
  apply (rule err_semilat_JType_esl, assumption+)
  done

lemma sl_triple_conv:
  "JVMType.sl G maxs maxr ==
  (states G maxs maxr, JVMType.le G maxs maxr, JVMType.sup G maxs maxr)"
  by (simp (no_asm) add: states_def JVMType.le_def JVMType.sup_def)

lemma is_type_pTs:
  "⟦ wf_prog wf_mb G; (C,S,fs,mdecls) ∈ set G; ((mn,pTs),rT,code) ∈ set mdecls ⟧
  ⟹ set pTs ⊆ types G"
```

```
proof
  assume "wf_prog wf_mb G"
         "(C,S,fs,mdecls) ∈ set G"
         "((mn,pTs),rT,code) ∈ set mdecls"
  hence "wf_mdecl wf_mb G C ((mn,pTs),rT,code)"
    by (rule wf_prog_wf_mdecl)
  hence "∀t ∈ set pTs. is_type G t"
    by (unfold wf_mdecl_def wf_mhead_def) auto
  moreover
  fix t assume "t ∈ set pTs"
  ultimately
  have "is_type G t" by blast
  thus "t ∈ types G" ..
qed


lemma jvm_prog_lift:
  assumes wf:
  "wf_prog (λG C bd. P G C bd) G"

  assumes rule:
  "⋀wf_mb C mn pTs C rT maxs maxl b et bd.
   wf_prog wf_mb G ⟹
   method (G,C) (mn,pTs) = Some (C,rT,maxs,maxl,b,et) ⟹
   is_class G C ⟹
   set pTs ⊆ types G ⟹
   bd = ((mn,pTs),rT,maxs,maxl,b,et) ⟹
   P G C bd ⟹
   Q G C bd"

  shows
  "wf_prog (λG C bd. Q G C bd) G"
proof -
  from wf show ?thesis
    apply (unfold wf_prog_def wf_cdecl_def)
    apply clarsimp
    apply (drule bspec, assumption)
    apply (unfold wf_cdecl_mdecl_def)
    apply clarsimp
    apply (drule bspec, assumption)
    apply (frule methd [OF wf [THEN wf_prog_ws_prog]], assumption+)
    apply (frule is_type_pTs [OF wf], assumption+)
    apply clarify
    apply (drule rule [OF wf], assumption+)
    apply (rule HOL.refl)
    apply assumption+
    done
qed

end
```

4.19 LBV for the JVM

theory *LBVJVM*
imports *Typing_Framework_JVM*
begin

```
types prog_cert = "cname ⇒ sig ⇒ JVMType.state list"

definition check_cert :: "jvm_prog ⇒ nat ⇒ nat ⇒ nat ⇒ JVMType.state list ⇒ bool"
where
  "check_cert G mxs mxr n cert ≡ check_types G mxs mxr cert ∧ length cert = n+1 ∧
                                (∀i<n. cert!i ≠ Err) ∧ cert!n = OK None"

definition lbvjvm :: "jvm_prog ⇒ nat ⇒ nat ⇒ ty ⇒ exception_table ⇒
             JVMType.state list ⇒ instr list ⇒ JVMType.state ⇒ JVMType.state" where
  "lbvjvm G maxs maxr rT et cert bs ≡
  wtl_inst_list bs cert  (JVMType.sup G maxs maxr) (JVMType.le G maxs maxr) Err (OK None)
(exec G maxs rT et bs) 0"

definition wt_lbv :: "jvm_prog ⇒ cname ⇒ ty list ⇒ ty ⇒ nat ⇒ nat ⇒
             exception_table ⇒ JVMType.state list ⇒ instr list ⇒ bool" where
  "wt_lbv G C pTs rT mxs mxl et cert ins ≡
   check_bounded ins et ∧
   check_cert G mxs (1+size pTs+mxl) (length ins) cert ∧
   0 < size ins ∧
   (let start  = Some ([],(OK (Class C))#((map OK pTs))@(replicate mxl Err));
        result = lbvjvm G mxs (1+size pTs+mxl) rT et cert ins (OK start)
    in result ≠ Err)"

definition wt_jvm_prog_lbv :: "jvm_prog ⇒ prog_cert ⇒ bool" where
  "wt_jvm_prog_lbv G cert ≡
  wf_prog (λG C (sig,rT,(maxs,maxl,b,et)). wt_lbv G C (snd sig) rT maxs maxl et (cert
C sig) b) G"

definition mk_cert :: "jvm_prog ⇒ nat ⇒ ty ⇒ exception_table ⇒ instr list
              ⇒ method_type ⇒ JVMType.state list" where
  "mk_cert G maxs rT et bs phi ≡ make_cert (exec G maxs rT et bs) (map OK phi) (OK None)"

definition prg_cert :: "jvm_prog ⇒ prog_type ⇒ prog_cert" where
  "prg_cert G phi C sig ≡ let (C,rT,(maxs,maxl,ins,et)) = the (method (G,C) sig) in
                          mk_cert G maxs rT et ins (phi C sig)"


lemma wt_method_def2:
  fixes pTs and mxl and G and mxs and rT and et and bs and phi
  defines [simp]: "mxr   ≡ 1 + length pTs + mxl"
  defines [simp]: "r     ≡ sup_state_opt G"
  defines [simp]: "app0  ≡ λpc. app (bs!pc) G mxs rT pc et"
  defines [simp]: "step0 ≡ λpc. eff (bs!pc) G pc et"

  shows
  "wt_method G C pTs rT mxs mxl bs et phi =
  (bs ≠ [] ∧
   length phi = length bs ∧
```

```
    check_bounded bs et ∧
    check_types G mxs mxr (map OK phi) ∧
    wt_start G C pTs mxl phi ∧
    wt_app_eff r app0 step0 phi)"
  by (auto simp add: wt_method_def wt_app_eff_def wt_instr_def lesub_def
           dest: check_bounded_is_bounded boundedD)


lemma check_certD:
  "check_cert G mxs mxr n cert ⟹ cert_ok cert n Err (OK None) (states G mxs mxr)"
  apply (unfold cert_ok_def check_cert_def check_types_def)
  apply (auto simp add: list_all_iff)
  done


lemma wt_lbv_wt_step:
  assumes wf:  "wf_prog wf_mb G"
  assumes lbv: "wt_lbv G C pTs rT mxs mxl et cert ins"
  assumes C:   "is_class G C"
  assumes pTs: "set pTs ⊆ types G"

  defines [simp]: "mxr ≡ 1+length pTs+mxl"

  shows "∃ts ∈ list (size ins) (states G mxs mxr).
            wt_step (JVMType.le G mxs mxr) Err (exec G mxs rT et ins) ts
          ∧ OK (Some ([],(OK (Class C))#((map OK pTs))@(replicate mxl Err))) <=_(JVMType.le
G mxs mxr) ts!0"
proof -
  let ?step = "exec G mxs rT et ins"
  let ?r    = "JVMType.le G mxs mxr"
  let ?f    = "JVMType.sup G mxs mxr"
  let ?A    = "states G mxs mxr"

  have "semilat (JVMType.sl G mxs mxr)"
    by (rule semilat_JVM_slI, rule wf_prog_ws_prog, rule wf)
  hence "semilat (?A, ?r, ?f)" by (unfold sl_triple_conv)
  moreover
  have "top ?r Err"  by (simp add: JVM_le_unfold)
  moreover
  have "Err ∈ ?A" by (simp add: JVM_states_unfold)
  moreover
  have "bottom ?r (OK None)"
    by (simp add: JVM_le_unfold bottom_def)
  moreover
  have "OK None ∈ ?A" by (simp add: JVM_states_unfold)
  moreover
  from lbv
  have "bounded ?step (length ins)"
    by (clarsimp simp add: wt_lbv_def exec_def)
       (intro bounded_lift check_bounded_is_bounded)
  moreover
  from lbv
  have "cert_ok cert (length ins) Err (OK None) ?A"
    by (unfold wt_lbv_def) (auto dest: check_certD)
```

```
  moreover
  from wf have "pres_type ?step (length ins) ?A" by (rule exec_pres_type)
  moreover
  let ?start = "OK (Some ([],(OK (Class C))#(map OK pTs)@(replicate mxl Err)))"
  from lbv
  have "wtl_inst_list ins cert ?f ?r Err (OK None) ?step 0 ?start ≠ Err"
    by (simp add: wt_lbv_def lbvjvm_def)
  moreover
  from C pTs have "?start ∈ ?A"
    by (unfold JVM_states_unfold) (auto intro: list_appendI, force)
  moreover
  from lbv have "0 < length ins" by (simp add: wt_lbv_def)
  ultimately
  show ?thesis by (rule lbvs.wtl_sound_strong [OF lbvs.intro, OF lbv.intro lbvs_axioms.intro,
OF Semilat.intro lbv_axioms.intro])
qed

lemma wt_lbv_wt_method:
  assumes wf:  "wf_prog wf_mb G"
  assumes lbv: "wt_lbv G C pTs rT mxs mxl et cert ins"
  assumes C:   "is_class G C"
  assumes pTs: "set pTs ⊆ types G"

  shows "∃phi. wt_method G C pTs rT mxs mxl ins et phi"
proof -
  let ?mxr   = "1 + length pTs + mxl"
  let ?step  = "exec G mxs rT et ins"
  let ?r     = "JVMType.le G mxs ?mxr"
  let ?f     = "JVMType.sup G mxs ?mxr"
  let ?A     = "states G mxs ?mxr"
  let ?start = "OK (Some ([],(OK (Class C))#(map OK pTs)@(replicate mxl Err)))"

  from lbv have l: "ins ≠ []" by (simp add: wt_lbv_def)
  moreover
  from wf lbv C pTs
  obtain phi where
    list:  "phi ∈ list (length ins) ?A" and
    step:  "wt_step ?r Err ?step phi" and
    start: "?start <=_?r phi!0"
    by (blast dest: wt_lbv_wt_step)
  from list have [simp]: "length phi = length ins" by simp
  have "length (map ok_val phi) = length ins" by simp
  moreover
  from l have 0: "0 < length phi" by simp
  with step obtain phi0 where "phi!0 = OK phi0"
    by (unfold wt_step_def) blast
  with start 0
  have "wt_start G C pTs mxl (map ok_val phi)"
    by (simp add: wt_start_def JVM_le_Err_conv lesub_def)
  moreover
  from lbv  have chk_bounded: "check_bounded ins et"
    by (simp add: wt_lbv_def)
  moreover {
    from list
```

```
    have "check_types G mxs ?mxr phi"
      by (simp add: check_types_def)
    also from step
    have [symmetric]: "map OK (map ok_val phi) = phi"
      by (auto intro!: nth_equalityI simp add: wt_step_def)
    finally have "check_types G mxs ?mxr (map OK (map ok_val phi))" .
  }
  moreover {
    let ?app = "λpc. app (ins!pc) G mxs rT pc et"
    let ?eff = "λpc. eff (ins!pc) G pc et"

    from chk_bounded
    have "bounded (err_step (length ins) ?app ?eff) (length ins)"
      by (blast dest: check_bounded_is_bounded boundedD intro: bounded_err_stepI)
    moreover
    from step
    have "wt_err_step (sup_state_opt G) ?step phi"
      by (simp add: wt_err_step_def JVM_le_Err_conv)
    ultimately
    have "wt_app_eff (sup_state_opt G) ?app ?eff (map ok_val phi)"
      by (auto intro: wt_err_imp_wt_app_eff simp add: exec_def)
  }
  ultimately
  have "wt_method G C pTs rT mxs mxl ins et (map ok_val phi)"
    by - (rule wt_method_def2 [THEN iffD2], simp)
  thus ?thesis ..
qed


lemma wt_method_wt_lbv:
  assumes wf:  "wf_prog wf_mb G"
  assumes wt:  "wt_method G C pTs rT mxs mxl ins et phi"
  assumes C:   "is_class G C"
  assumes pTs: "set pTs ⊆ types G"

  defines [simp]: "cert ≡ mk_cert G mxs rT et ins phi"

  shows "wt_lbv G C pTs rT mxs mxl et cert ins"
proof -
  let ?mxr  = "1 + length pTs + mxl"
  let ?step = "exec G mxs rT et ins"
  let ?app  = "λpc. app (ins!pc) G mxs rT pc et"
  let ?eff  = "λpc. eff (ins!pc) G pc et"
  let ?r    = "JVMType.le G mxs ?mxr"
  let ?f    = "JVMType.sup G mxs ?mxr"
  let ?A    = "states G mxs ?mxr"
  let ?phi  = "map OK phi"
  let ?cert = "make_cert ?step ?phi (OK None)"

  from wt have
    0:          "0 < length ins" and
    length:     "length ins = length ?phi" and
    ck_bounded: "check_bounded ins et" and
    ck_types:   "check_types G mxs ?mxr ?phi" and
```

```
    wt_start:   "wt_start G C pTs mxl phi" and
    app_eff:    "wt_app_eff (sup_state_opt G) ?app ?eff phi"
    by (simp_all add: wt_method_def2)

  have "semilat (JVMType.sl G mxs ?mxr)"
    by (rule semilat_JVM_slI) (rule wf_prog_ws_prog [OF wf])
  hence "semilat (?A, ?r, ?f)" by (unfold sl_triple_conv)
  moreover
  have "top ?r Err"  by (simp add: JVM_le_unfold)
  moreover
  have "Err ∈ ?A" by (simp add: JVM_states_unfold)
  moreover
  have "bottom ?r (OK None)"
    by (simp add: JVM_le_unfold bottom_def)
  moreover
  have "OK None ∈ ?A" by (simp add: JVM_states_unfold)
  moreover
  from ck_bounded
  have bounded: "bounded ?step (length ins)"
    by (clarsimp simp add: exec_def)
       (intro bounded_lift check_bounded_is_bounded)
  with wf
  have "mono ?r ?step (length ins) ?A"
    by (rule wf_prog_ws_prog [THEN exec_mono])
  hence "mono ?r ?step (length ?phi) ?A" by (simp add: length)
  moreover
  from wf have "pres_type ?step (length ins) ?A" by (rule exec_pres_type)
  hence "pres_type ?step (length ?phi) ?A" by (simp add: length)
  moreover
  from ck_types
  have "set ?phi ⊆ ?A" by (simp add: check_types_def)
  hence "∀pc. pc < length ?phi ⟶ ?phi!pc ∈ ?A ∧ ?phi!pc ≠ Err" by auto
  moreover
  from bounded
  have "bounded (exec G mxs rT et ins) (length ?phi)" by (simp add: length)
  moreover
  have "OK None ≠ Err" by simp
  moreover
  from bounded length app_eff
  have "wt_err_step (sup_state_opt G) ?step ?phi"
    by (auto intro: wt_app_eff_imp_wt_err simp add: exec_def)
  hence "wt_step ?r Err ?step ?phi"
    by (simp add: wt_err_step_def JVM_le_Err_conv)
  moreover
  let ?start = "OK (Some ([],(OK (Class C))#(map OK pTs)@(replicate mxl Err)))"
  from 0 length have "0 < length phi" by auto
  hence "?phi!0 = OK (phi!0)" by simp
  with wt_start have "?start <=_?r ?phi!0"
    by (clarsimp simp add: wt_start_def lesub_def JVM_le_Err_conv)
  moreover
  from C pTs have "?start ∈ ?A"
    by (unfold JVM_states_unfold) (auto intro: list_appendI, force)
  moreover
  have "?start ≠ Err" by simp
```

```
  moreover
  note length
  ultimately
  have "wtl_inst_list ins ?cert ?f ?r Err (OK None) ?step 0 ?start ≠ Err"
    by (rule lbvc.wtl_complete [OF lbvc.intro, OF lbv.intro lbvc_axioms.intro, OF Semilat.intro
lbv_axioms.intro])
  moreover
  from 0 length have "phi ≠ []" by auto
  moreover
  from ck_types
  have "check_types G mxs ?mxr ?cert"
    by (auto simp add: make_cert_def check_types_def JVM_states_unfold)
  moreover
  note ck_bounded 0 length
  ultimately
  show ?thesis
    by (simp add: wt_lbv_def lbvjvm_def mk_cert_def
      check_cert_def make_cert_def nth_append)
qed



theorem jvm_lbv_correct:
  "wt_jvm_prog_lbv G Cert ⟹ ∃Phi. wt_jvm_prog G Phi"
proof -
  let ?Phi = "λC sig. let (C,rT,(maxs,maxl,ins,et)) = the (method (G,C) sig) in
              SOME phi. wt_method G C (snd sig) rT maxs maxl ins et phi"

  assume "wt_jvm_prog_lbv G Cert"
  hence "wt_jvm_prog G ?Phi"
    apply (unfold wt_jvm_prog_def wt_jvm_prog_lbv_def)
    apply (erule jvm_prog_lift)
    apply (auto dest: wt_lbv_wt_method intro: someI)
    done
  thus ?thesis by blast
qed

theorem jvm_lbv_complete:
  "wt_jvm_prog G Phi ⟹ wt_jvm_prog_lbv G (prg_cert G Phi)"
  apply (unfold wt_jvm_prog_def wt_jvm_prog_lbv_def)
  apply (erule jvm_prog_lift)
  apply (auto simp add: prg_cert_def intro: wt_method_wt_lbv)
  done

end
```

4.20 BV Type Safety Invariant

```
theory Correct
imports BVSpec "../JVM/JVMExec"
begin

definition approx_val :: "[jvm_prog,aheap,val,ty err] ⇒ bool" where
  "approx_val G h v any == case any of Err ⇒ True | OK T ⇒ G,h⊢v::≼T"

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

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

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

primrec correct_frames  :: "[jvm_prog,aheap,prog_type,ty,sig,frame list] ⇒ bool" where
  "correct_frames G hp phi rT0 sig0 [] = True"
| "correct_frames G hp phi rT0 sig0 (f#frs) =
    (let (stk,loc,C,sig,pc) = f in
    (∃ST LT rT maxs maxl ins et.
      phi C sig ! pc = Some (ST,LT) ∧ is_class G C ∧
      method (G,C) sig = Some(C,rT,(maxs,maxl,ins,et)) ∧
    (∃C' mn pTs. ins!pc = (Invoke C' mn pTs) ∧
           (mn,pTs) = sig0 ∧
           (∃apTs D ST' LT'.
           (phi C sig)!pc = Some ((rev apTs) @ (Class D) # ST', LT') ∧
           length apTs = length pTs ∧
           (∃D' rT' maxs' maxl' ins' et'.
             method (G,D) sig0 = Some(D',rT',(maxs',maxl',ins',et')) ∧
             G ⊢ rT0 ≼ rT') ∧
     correct_frame G hp (ST, LT) maxl ins f ∧
     correct_frames G hp phi rT sig frs))))"

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

```
  | Some x ⇒ frs = []"
```

notation *(xsymbols)*

```
 correct_state  ("_,_ ⊢JVM _ √"  [51,51] 50)
```

lemma *sup_ty_opt_OK:*

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

4.20.1 approx-val

```
lemma approx_val_Err [simp,intro!]:
  "approx_val G hp x Err"
  by (simp add: approx_val_def)

lemma approx_val_OK [iff]:
  "approx_val G hp x (OK T) = (G,hp ⊢ x ::≼ T)"
  by (simp add: approx_val_def)

lemma approx_val_Null [simp,intro!]:
  "approx_val G hp Null (OK (RefT x))"
  by (auto simp add: approx_val_def)

lemma approx_val_sup_heap:
  "⟦ approx_val G hp v T; hp ≤| hp' ⟧ ⟹ approx_val G hp' v T"
  by (cases T) (blast intro: conf_hext)+

lemma approx_val_heap_update:
  "⟦ hp a = Some obj'; G,hp⊢ v::≼T; obj_ty obj = obj_ty obj'⟧
  ⟹ G,hp(a↦obj)⊢ v::≼T"
  by (cases v, auto simp add: obj_ty_def conf_def)

lemma approx_val_widen:
  "⟦ approx_val G hp v T; G ⊢ T <=o T'; wf_prog wt G ⟧
  ⟹ approx_val G hp v T'"
  by (cases T', auto simp add: sup_ty_opt_OK intro: conf_widen)
```

4.20.2 approx-loc

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

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

lemma approx_loc_nth:
  "⟦ approx_loc G hp loc LT; n < length LT ⟧
```

```
  ⟹ approx_val G hp (loc!n) (LT!n)"
  by (simp add: approx_loc_def list_all2_conv_all_nth)

lemma approx_loc_imp_approx_val_sup:
  "⟦approx_loc G hp loc LT; n < length LT; LT ! n = OK T; G ⊢ T ≼ T'; wf_prog wt G⟧
  ⟹ G,hp ⊢ (loc!n) ::≼ T'"
  apply (drule approx_loc_nth, assumption)
  apply simp
  apply (erule conf_widen, assumption+)
  done

lemma approx_loc_conv_all_nth:
  "approx_loc G hp loc LT =
  (length loc = length LT ∧ (∀n < length loc. approx_val G hp (loc!n) (LT!n)))"
  by (simp add: approx_loc_def list_all2_conv_all_nth)

lemma approx_loc_sup_heap:
  "⟦ approx_loc G hp loc LT; hp ≤| hp' ⟧
  ⟹ approx_loc G hp' loc LT"
  apply (clarsimp simp add: approx_loc_conv_all_nth)
  apply (blast intro: approx_val_sup_heap)
  done

lemma approx_loc_widen:
  "⟦ approx_loc G hp loc LT; G ⊢ LT <=l LT'; wf_prog wt G ⟧
  ⟹ approx_loc G hp loc LT'"
apply (unfold Listn.le_def lesub_def sup_loc_def)
apply (simp (no_asm_use) only: list_all2_conv_all_nth approx_loc_conv_all_nth)
apply (simp (no_asm_simp))
apply clarify
apply (erule allE, erule impE)
 apply simp
apply (erule approx_val_widen)
 apply simp
apply assumption
done

lemma loc_widen_Err [dest]:
  "⋀XT. G ⊢ replicate n Err <=l XT ⟹ XT = replicate n Err"
  by (induct n) auto

lemma approx_loc_Err [iff]:
  "approx_loc G hp (replicate n v) (replicate n Err)"
  by (induct n) auto

lemma approx_loc_subst:
  "⟦ approx_loc G hp loc LT; approx_val G hp x X ⟧
  ⟹ approx_loc G hp (loc[idx:=x]) (LT[idx:=X])"
apply (unfold approx_loc_def list_all2_def)
apply (auto dest: subsetD [OF set_update_subset_insert] simp add: zip_update)
done

lemma approx_loc_append:
  "length l1=length L1 ⟹
```

```
  approx_loc G hp (l1@l2) (L1@L2) =
  (approx_loc G hp l1 L1 ∧ approx_loc G hp l2 L2)"
  apply (unfold approx_loc_def list_all2_def)
  apply (simp cong: conj_cong)
  apply blast
  done
```

4.20.3 approx-stk

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

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

lemma approx_stk_sup_heap:
  "⟦ approx_stk G hp stk ST; hp ≤| hp' ⟧ ⟹ approx_stk G hp' stk ST"
  by (auto intro: approx_loc_sup_heap simp add: approx_stk_def)

lemma approx_stk_widen:
  "⟦ approx_stk G hp stk ST; G ⊢ map OK ST <=l map OK ST'; wf_prog wt G ⟧
  ⟹ approx_stk G hp stk ST'"
  by (auto elim: approx_loc_widen simp add: approx_stk_def)

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

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

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

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

lemma approx_stk_all_widen:
  "⟦ approx_stk G hp stk ST; ∀(x, y) ∈ set (zip ST ST'). G ⊢ x ≼ y; length ST = length
ST'; wf_prog wt G ⟧
  ⟹ approx_stk G hp stk ST'"
apply (unfold approx_stk_def)
apply (clarsimp simp add: approx_loc_conv_all_nth all_set_conv_all_nth)
```

```
apply (erule allE, erule impE, assumption)
apply (erule allE, erule impE, assumption)
apply (erule conf_widen, assumption+)
done
```

4.20.4 oconf

```
lemma oconf_field_update:
  "⟦map_of (fields (G, oT)) FD = Some T; G,hp⊢v::≼T; G,hp⊢(oT,fs)√ ⟧
  ⟹ G,hp⊢(oT, fs(FD↦v))√"
  by (simp add: oconf_def lconf_def)

lemma oconf_newref:
  "⟦hp oref = None; G,hp ⊢ obj √; G,hp ⊢ obj' √⟧ ⟹ G,hp(oref↦obj') ⊢ obj √"
  apply (unfold oconf_def lconf_def)
  apply simp
  apply (blast intro: conf_hext hext_new)
  done

lemma oconf_heap_update:
  "⟦ hp a = Some obj'; obj_ty obj' = obj_ty obj''; G,hp⊢obj√ ⟧
  ⟹ G,hp(a↦obj'')⊢obj√"
  apply (unfold oconf_def lconf_def)
  apply (fastsimp intro: approx_val_heap_update)
  done
```

4.20.5 hconf

```
lemma hconf_newref:
  "⟦ hp oref = None; G⊢h hp√; G,hp⊢obj√ ⟧ ⟹ G⊢h hp(oref↦obj)√"
  apply (simp add: hconf_def)
  apply (fast intro: oconf_newref)
  done

lemma hconf_field_update:
  "⟦ map_of (fields (G, oT)) X = Some T; hp a = Some(oT,fs);
     G,hp⊢v::≼T; G⊢h hp√ ⟧
  ⟹ G⊢h hp(a ↦ (oT, fs(X↦v)))√"
  apply (simp add: hconf_def)
  apply (fastsimp intro: oconf_heap_update oconf_field_update
                  simp add: obj_ty_def)
  done
```

4.20.6 preallocated

```
lemma preallocated_field_update:
  "⟦ map_of (fields (G, oT)) X = Some T; hp a = Some(oT,fs);
     G⊢h hp√; preallocated hp ⟧
  ⟹ preallocated (hp(a ↦ (oT, fs(X↦v))))"
  apply (unfold preallocated_def)
  apply (rule allI)
  apply (erule_tac x=x in allE)
  apply simp
  apply (rule ccontr)
  apply (unfold hconf_def)
```

```
  apply (erule allE, erule allE, erule impE, assumption)
  apply (unfold oconf_def lconf_def)
  apply (simp del: split_paired_All)
  done


lemma
  assumes none: "hp oref = None" and alloc: "preallocated hp"
  shows preallocated_newref: "preallocated (hp(oref↦obj))"
proof (cases oref)
  case (XcptRef x)
  with none alloc have "False" by (auto elim: preallocatedE [of _ x])
  thus ?thesis ..
next
  case (Loc l)
  with alloc show ?thesis by (simp add: preallocated_def)
qed
```

4.20.7 correct-frames

```
lemmas [simp del] = fun_upd_apply

lemma correct_frames_field_update [rule_format]:
  "∀rT C sig.
  correct_frames G hp phi rT sig frs ⟶
  hp a = Some (C,fs) ⟶
  map_of (fields (G, C)) fl = Some fd ⟶
  G,hp⊢v::≼fd
  ⟶ correct_frames G (hp(a ↦ (C, fs(fl↦v)))) phi rT sig frs"
apply (induct frs)
 apply simp
apply clarify
apply (simp (no_asm_use))
apply clarify
apply (unfold correct_frame_def)
apply (simp (no_asm_use))
apply clarify
apply (intro exI conjI)
    apply assumption+
   apply (erule approx_stk_sup_heap)
   apply (erule hext_upd_obj)
  apply (erule approx_loc_sup_heap)
  apply (erule hext_upd_obj)
 apply assumption+
apply blast
done

lemma correct_frames_newref [rule_format]:
  "∀rT C sig.
  hp x = None ⟶
  correct_frames G hp phi rT sig frs ⟶
  correct_frames G (hp(x ↦ obj)) phi rT sig frs"
apply (induct frs)
 apply simp
```

```
apply clarify
apply (simp (no_asm_use))
apply clarify
apply (unfold correct_frame_def)
apply (simp (no_asm_use))
apply clarify
apply (intro exI conjI)
    apply assumption+
   apply (erule approx_stk_sup_heap)
   apply (erule hext_new)
  apply (erule approx_loc_sup_heap)
  apply (erule hext_new)
 apply assumption+
apply blast
done

end
```

4.21 BV Type Safety Proof

```
theory BVSpecTypeSafe
imports Correct
begin
```

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

4.21.1 Preliminaries

Simp and intro setup for the type safety proof:

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

lemmas widen_rules[intro] = approx_val_widen approx_loc_widen approx_stk_widen

lemmas [simp del] = split_paired_All
```

If we have a welltyped program and a conforming state, we can directly infer that the current instruction is well typed:

```
lemma wt_jvm_prog_impl_wt_instr_cor:
  "⟦ wt_jvm_prog G phi; method (G,C) sig = Some (C,rT,maxs,maxl,ins,et);
      G,phi ⊢JVM (None, hp, (stk,loc,C,sig,pc)#frs)√ ⟧
  ⟹ wt_instr (ins!pc) G rT (phi C sig) maxs (length ins) et pc"
apply (unfold correct_state_def Let_def correct_frame_def)
apply simp
apply (blast intro: wt_jvm_prog_impl_wt_instr)
done
```

4.21.2 Exception Handling

Exceptions don't touch anything except the stack:

```
lemma exec_instr_xcpt:
  "(fst (exec_instr i G hp stk vars Cl sig pc frs) = Some xcp)
  = (∃stk'. exec_instr i G hp stk vars Cl sig pc frs =
           (Some xcp, hp, (stk', vars, Cl, sig, pc)#frs))"
  by (cases i, auto simp add: split_beta split: split_if_asm)
```

Relates `match_any` from the Bytecode Verifier with `match_exception_table` from the operational semantics:

```
lemma in_match_any:
  "match_exception_table G xcpt pc et = Some pc' ⟹
  ∃C. C ∈ set (match_any G pc et) ∧ G ⊢ xcpt ≼C C ∧
      match_exception_table G C pc et = Some pc'"
  (is "PROP ?P et" is "?match et ⟹ ?match_any et")
proof (induct et)
  show "PROP ?P []"
    by simp

  fix e es
  assume IH: "PROP ?P es"
```

```
  assume match: "?match (e#es)"

  obtain start_pc end_pc handler_pc catch_type where
    e [simp]: "e = (start_pc, end_pc, handler_pc, catch_type)"
    by (cases e)

  from IH match
  show "?match_any (e#es)"
  proof (cases "match_exception_entry G xcpt pc e")
    case False
    with match
    have "match_exception_table G xcpt pc es = Some pc'" by simp
    with IH
    obtain C where
      set: "C ∈ set (match_any G pc es)" and
      C:   "G ⊢ xcpt ≼C C" and
      m:   "match_exception_table G C pc es = Some pc'" by blast

    from set
    have "C ∈ set (match_any G pc (e#es))" by simp
    moreover
    from False C
    have "¬ match_exception_entry G C pc e"
      by - (erule contrapos_nn,
            auto simp add: match_exception_entry_def)
    with m
    have "match_exception_table G C pc (e#es) = Some pc'" by simp
    moreover note C
    ultimately
    show ?thesis by blast
  next
    case True with match
    have "match_exception_entry G catch_type pc e"
      by (simp add: match_exception_entry_def)
    moreover
    from True match
    obtain
      "start_pc ≤ pc"
      "pc < end_pc"
      "G ⊢ xcpt ≼C catch_type"
      "handler_pc = pc'"
      by (simp add: match_exception_entry_def)
    ultimately
    show ?thesis by auto
  qed
qed

lemma match_et_imp_match:
  "match_exception_table G (Xcpt X) pc et = Some handler
  ⟹ match G X pc et = [Xcpt X]"
  apply (simp add: match_some_entry)
  apply (induct et)
  apply (auto split: split_if_asm)
  done
```

We can prove separately that the recursive search for exception handlers (*find_handler*) in the frame stack results in a conforming state (if there was no matching exception handler in the current frame). We require that the exception is a valid heap address, and that the state before the exception occured conforms.

```
lemma uncaught_xcpt_correct:
  "⋀f. ⟦ wt_jvm_prog G phi; xcp = Addr adr; hp adr = Some T;
      G,phi ⊢JVM (None, hp, f#frs)√ ⟧
  ⟹ G,phi ⊢JVM (find_handler G (Some xcp) hp frs)√"
  (is "⋀f. ⟦ ?wt; ?adr; ?hp; ?correct (None, hp, f#frs) ⟧ ⟹ ?correct (?find frs)")
proof (induct frs)
  — the base case is trivial, as it should be
  show "?correct (?find [])" by (simp add: correct_state_def)

  — we will need both forms wt_jvm_prog and wf_prog later
  assume wt: ?wt
  then obtain mb where wf: "wf_prog mb G" by (simp add: wt_jvm_prog_def)

  — these two don't change in the induction:
  assume adr: ?adr
  assume hp: ?hp

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

  — the induction hypothesis as produced by Isabelle, immediatly simplified with the fixed assumptions
above
  assume "⋀f. ⟦ ?wt; ?adr; ?hp; ?correct (None, hp, f#frs') ⟧ ⟹ ?correct (?find frs')"

  with wt adr hp
  have IH: "⋀f. ?correct (None, hp, f#frs') ⟹ ?correct (?find frs')" by blast

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

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

  from cr
  obtain rT maxs maxl ins et where
    meth: "method (G,C) sig = Some (C,rT,maxs,maxl,ins,et)"
    by (simp add: correct_state_def, blast)

  hence [simp]: "ex_table_of (snd (snd (the (method (G, C) sig)))) = et"
    by simp

  show "?correct (?find (f'#frs'))"
  proof (cases "match_exception_table G (cname_of hp xcp) pc et")
    case None
    with cr' IH
    show ?thesis by simp
  next
    fix handler_pc
```

```
assume match: "match_exception_table G (cname_of hp xcp) pc et = Some handler_pc"
(is "?match (cname_of hp xcp) = _")

from wt meth cr' [simplified]
have wti: "wt_instr (ins!pc) G rT (phi C sig) maxs (length ins) et pc"
  by (rule wt_jvm_prog_impl_wt_instr_cor)

from cr meth
obtain C' mn pts ST LT where
  ins: "ins!pc = Invoke C' mn pts" (is "_ = ?i") and
  phi: "phi C sig ! pc = Some (ST, LT)"
  by (simp add: correct_state_def) blast

from match
obtain D where
  in_any: "D ∈ set (match_any G pc et)" and
  D:      "G ⊢ cname_of hp xcp ≼C D" and
  match': "?match D = Some handler_pc"
  by (blast dest: in_match_any)

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

from cr' phi meth f'
have "correct_frame G hp (ST, LT) maxl ins f'"
  by (unfold correct_state_def) auto
then obtain
 len: "length loc = 1+length (snd sig)+maxl" and
 loc: "approx_loc G hp loc LT"
  by (unfold correct_frame_def) auto

let ?f = "([xcp], loc, C, sig, handler_pc)"
have "correct_frame G hp (ST', LT') maxl ins ?f"
proof -
  from wf less loc
  have "approx_loc G hp loc LT'" by (simp add: sup_state_conv) blast
  moreover
  from D adr hp
  have "G,hp ⊢ xcp ::≼ Class D" by (simp add: conf_def obj_ty_def)
  with wf less loc
  have "approx_stk G hp [xcp] ST'"
    by (auto simp add: sup_state_conv approx_stk_def approx_val_def
             elim: conf_widen split: Err.split)
  moreover
```

```
      note len pc
      ultimately
      show ?thesis by (simp add: correct_frame_def)
    qed

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

declare raise_if_def [simp]
```

The requirement of lemma `uncaught_xcpt_correct` (that the exception is a valid reference on the heap) is always met for welltyped instructions and conformant states:

```
lemma exec_instr_xcpt_hp:
  "⟦  fst (exec_instr (ins!pc) G hp stk vars Cl sig pc frs) = Some xcp;
      wt_instr (ins!pc) G rT (phi C sig) maxs (length ins) et pc;
      G,phi ⊢JVM (None, hp, (stk,loc,C,sig,pc)#frs)√ ⟧
  ⟹ ∃adr T. xcp = Addr adr ∧ hp adr = Some T"
  (is "⟦ ?xcpt; ?wt; ?correct ⟧ ⟹ ?thesis")
proof -
  note [simp] = split_beta raise_system_xcpt_def
  note [split] = split_if_asm option.split_asm

  assume wt: ?wt ?correct
  hence pre: "preallocated hp" by (simp add: correct_state_def)

  assume xcpt: ?xcpt with pre show ?thesis
  proof (cases "ins!pc")
    case New with xcpt pre
    show ?thesis by (auto dest: new_Addr_OutOfMemory dest!: preallocatedD)
  next
    case Throw with xcpt wt
    show ?thesis
      by (auto simp add: wt_instr_def correct_state_def correct_frame_def
               dest: non_npD dest!: preallocatedD)
  qed (auto dest!: preallocatedD)
qed


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

Finally we can state that, whenever an exception occurs, the resulting next state always conforms:

```
lemma xcpt_correct:
  "⟦ wt_jvm_prog G phi;
     method (G,C) sig = Some (C,rT,maxs,maxl,ins,et);
     wt_instr (ins!pc) G rT (phi C sig) maxs (length ins) et pc;
     fst (exec_instr (ins!pc) G hp stk loc C sig pc frs) = Some xcp;
     Some state' = exec (G, None, hp, (stk,loc,C,sig,pc)#frs);
     G,phi ⊢JVM (None, hp, (stk,loc,C,sig,pc)#frs)√ ⟧
  ⟹ G,phi ⊢JVM state'√"
```

```
proof -
  assume wtp: "wt_jvm_prog G phi"
  assume meth: "method (G,C) sig = Some (C,rT,maxs,maxl,ins,et)"
  assume wt: "wt_instr (ins!pc) G rT (phi C sig) maxs (length ins) et pc"
  assume xp: "fst (exec_instr (ins!pc) G hp stk loc C sig pc frs) = Some xcp"
  assume s': "Some state' = exec (G, None, hp, (stk,loc,C,sig,pc)#frs)"
  assume correct: "G,phi ⊢JVM (None, hp, (stk,loc,C,sig,pc)#frs)√"

  from wtp obtain wfmb where wf: "wf_prog wfmb G" by (simp add: wt_jvm_prog_def)

  note xp' = meth s' xp

  note wtp
  moreover
  from xp wt correct
  obtain adr T where
    adr: "xcp = Addr adr" "hp adr = Some T"
    by (blast dest: exec_instr_xcpt_hp)
  moreover
  note correct
  ultimately
  have "G,phi ⊢JVM find_handler G (Some xcp) hp frs √" by (rule uncaught_xcpt_correct)
  with xp'
  have "match_exception_table G (cname_of hp xcp) pc et = None ⟹ ?thesis"
    (is "?m (cname_of hp xcp) = _ ⟹ _" is "?match = _ ⟹ _")
    by (clarsimp simp add: exec_instr_xcpt split_beta)
  moreover
  { fix handler
    assume some_handler: "?match = Some handler"

    from correct meth
    obtain ST LT where
      hp_ok:  "G ⊢h hp √" and
      prehp:  "preallocated hp" and
      "class":  "is_class G C" and
      phi_pc: "phi C sig ! pc = Some (ST, LT)" and
      frame:  "correct_frame G hp (ST, LT) maxl ins (stk, loc, C, sig, pc)" and
      frames: "correct_frames G hp phi rT sig frs"
      by (unfold correct_state_def) auto

    from frame obtain
      stk: "approx_stk G hp stk ST" and
      loc: "approx_loc G hp loc LT" and
      pc:  "pc < length ins" and
      len: "length loc = 1+length (snd sig)+maxl"
      by (unfold correct_frame_def) auto

    from wt obtain
      eff: "∀(pc', s')∈set (xcpt_eff (ins!pc) G pc (phi C sig!pc) et).
             pc' < length ins ∧ G ⊢ s' <=' phi C sig!pc'"
      by (simp add: wt_instr_def eff_def)

    from some_handler xp'
    have state':
```

```
      "state' = (None, hp, ([xcp], loc, C, sig, handler)#frs)"
      by (cases "ins!pc", auto simp add: raise_system_xcpt_def split_beta
                           split: split_if_asm)

    let ?f' = "([xcp], loc, C, sig, handler)"
    from eff
    obtain ST' LT' where
      phi_pc': "phi C sig ! handler = Some (ST', LT')" and
      frame': "correct_frame G hp (ST',LT') maxl ins ?f'"
    proof (cases "ins!pc")
      case Return — can't generate exceptions:
      with xp' have False by (simp add: split_beta split: split_if_asm)
      thus ?thesis ..
    next
      case New
      with some_handler xp'
      have xcp: "xcp = Addr (XcptRef OutOfMemory)"
        by (simp add: raise_system_xcpt_def split_beta new_Addr_OutOfMemory)
      with prehp have "cname_of hp xcp = Xcpt OutOfMemory" ..
      with New some_handler phi_pc eff
      obtain ST' LT' where
        phi': "phi C sig ! handler = Some (ST', LT')" and
        less: "G ⊢ ([Class (Xcpt OutOfMemory)], LT) <=s (ST', LT')" and
        pc':  "handler < length ins"
        by (simp add: xcpt_eff_def match_et_imp_match) blast
      note phi'
      moreover
      { from xcp prehp
        have "G,hp ⊢ xcp ::≼ Class (Xcpt OutOfMemory)"
          by (auto simp add: conf_def obj_ty_def dest!: preallocatedD)
        moreover
        from wf less loc
        have "approx_loc G hp loc LT'"
          by (simp add: sup_state_conv) blast
        moreover
        note wf less pc' len
        ultimately
        have "correct_frame G hp (ST',LT') maxl ins ?f'"
          by (unfold correct_frame_def) (auto simp add: sup_state_conv
              approx_stk_def approx_val_def split: err.split elim: conf_widen)
      }
      ultimately
      show ?thesis by (rule that)
    next
      case Getfield
      with some_handler xp'
      have xcp: "xcp = Addr (XcptRef NullPointer)"
        by (simp add: raise_system_xcpt_def split_beta split: split_if_asm)
      with prehp have "cname_of hp xcp = Xcpt NullPointer" ..
      with Getfield some_handler phi_pc eff
      obtain ST' LT' where
        phi': "phi C sig ! handler = Some (ST', LT')" and
        less: "G ⊢ ([Class (Xcpt NullPointer)], LT) <=s (ST', LT')" and
        pc':  "handler < length ins"
```

```
      by (simp add: xcpt_eff_def match_et_imp_match) blast
    note phi'
    moreover
    { from xcp prehp
      have "G,hp ⊢ xcp ::≼ Class (Xcpt NullPointer)"
        by (auto simp add: conf_def obj_ty_def dest!: preallocatedD)
      moreover
      from wf less loc
      have "approx_loc G hp loc LT'"
        by (simp add: sup_state_conv) blast
      moreover
      note wf less pc' len
      ultimately
      have "correct_frame G hp (ST',LT') maxl ins ?f'"
        by (unfold correct_frame_def) (auto simp add: sup_state_conv
            approx_stk_def approx_val_def split: err.split elim: conf_widen)
    }
    ultimately
    show ?thesis by (rule that)
  next
    case Putfield
    with some_handler xp'
    have xcp: "xcp = Addr (XcptRef NullPointer)"
      by (simp add: raise_system_xcpt_def split_beta split: split_if_asm)
    with prehp have "cname_of hp xcp = Xcpt NullPointer" ..
    with Putfield some_handler phi_pc eff
    obtain ST' LT' where
      phi': "phi C sig ! handler = Some (ST', LT')" and
      less: "G ⊢ ([Class (Xcpt NullPointer)], LT) <=s (ST', LT')" and
      pc':  "handler < length ins"
      by (simp add: xcpt_eff_def match_et_imp_match) blast
    note phi'
    moreover
    { from xcp prehp
      have "G,hp ⊢ xcp ::≼ Class (Xcpt NullPointer)"
        by (auto simp add: conf_def obj_ty_def dest!: preallocatedD)
      moreover
      from wf less loc
      have "approx_loc G hp loc LT'"
        by (simp add: sup_state_conv) blast
      moreover
      note wf less pc' len
      ultimately
      have "correct_frame G hp (ST',LT') maxl ins ?f'"
        by (unfold correct_frame_def) (auto simp add: sup_state_conv
            approx_stk_def approx_val_def split: err.split elim: conf_widen)
    }
    ultimately
    show ?thesis by (rule that)
  next
    case Checkcast
    with some_handler xp'
    have xcp: "xcp = Addr (XcptRef ClassCast)"
      by (simp add: raise_system_xcpt_def split_beta split: split_if_asm)
```

```
    with prehp have "cname_of hp xcp = Xcpt ClassCast" ..
    with Checkcast some_handler phi_pc eff
    obtain ST' LT' where
      phi': "phi C sig ! handler = Some (ST', LT')" and
      less: "G ⊢ ([Class (Xcpt ClassCast)], LT) <=s (ST', LT')" and
      pc':  "handler < length ins"
      by (simp add: xcpt_eff_def match_et_imp_match) blast
    note phi'
    moreover
    { from xcp prehp
      have "G,hp ⊢ xcp ::≼ Class (Xcpt ClassCast)"
        by (auto simp add: conf_def obj_ty_def dest!: preallocatedD)
      moreover
      from wf less loc
      have "approx_loc G hp loc LT'"
        by (simp add: sup_state_conv) blast
      moreover
      note wf less pc' len
      ultimately
      have "correct_frame G hp (ST',LT') maxl ins ?f'"
        by (unfold correct_frame_def) (auto simp add: sup_state_conv
            approx_stk_def approx_val_def split: err.split elim: conf_widen)
    }
    ultimately
    show ?thesis by (rule that)
  next
    case Invoke
    with phi_pc eff
    have
      "∀D∈set (match_any G pc et).
      the (?m D) < length ins ∧ G ⊢ Some ([Class D], LT) <=' phi C sig!the (?m D)"
      by (simp add: xcpt_eff_def)
    moreover
    from some_handler
    obtain D where
      "D ∈ set (match_any G pc et)" and
      D: "G ⊢ cname_of hp xcp ≼C D" and
      "?m D = Some handler"
      by (blast dest: in_match_any)
    ultimately
    obtain
      pc': "handler < length ins" and
      "G ⊢ Some ([Class D], LT) <=' phi C sig ! handler"
      by auto
    then
    obtain ST' LT' where
      phi': "phi C sig ! handler = Some (ST', LT')" and
      less: "G ⊢ ([Class D], LT) <=s (ST', LT')"
      by auto
    from xp wt correct
    obtain addr T where
      xcp: "xcp = Addr addr" "hp addr = Some T"
      by (blast dest: exec_instr_xcpt_hp)
    note phi'
```

```
    moreover
    { from xcp D
      have "G,hp ⊢ xcp ::≼ Class D"
        by (simp add: conf_def obj_ty_def)
      moreover
      from wf less loc
      have "approx_loc G hp loc LT'"
        by (simp add: sup_state_conv) blast
      moreover
      note wf less pc' len
      ultimately
      have "correct_frame G hp (ST',LT') maxl ins ?f'"
        by (unfold correct_frame_def) (auto simp add: sup_state_conv
            approx_stk_def approx_val_def split: err.split elim: conf_widen)
    }
    ultimately
    show ?thesis by (rule that)
  next
    case Throw
    with phi_pc eff
    have
      "∀D∈set (match_any G pc et).
      the (?m D) < length ins ∧ G ⊢ Some ([Class D], LT) <=' phi C sig!the (?m D)"
      by (simp add: xcpt_eff_def)
    moreover
    from some_handler
    obtain D where
      "D ∈ set (match_any G pc et)" and
      D: "G ⊢ cname_of hp xcp ≼C D" and
      "?m D = Some handler"
      by (blast dest: in_match_any)
    ultimately
    obtain
      pc': "handler < length ins" and
      "G ⊢ Some ([Class D], LT) <=' phi C sig ! handler"
      by auto
    then
    obtain ST' LT' where
      phi': "phi C sig ! handler = Some (ST', LT')" and
      less: "G ⊢ ([Class D], LT) <=s (ST', LT')"
      by auto
    from xp wt correct
    obtain addr T where
      xcp: "xcp = Addr addr" "hp addr = Some T"
      by (blast dest: exec_instr_xcpt_hp)
    note phi'
    moreover
    { from xcp D
      have "G,hp ⊢ xcp ::≼ Class D"
        by (simp add: conf_def obj_ty_def)
      moreover
      from wf less loc
      have "approx_loc G hp loc LT'"
        by (simp add: sup_state_conv) blast
```

```
        moreover
        note wf less pc' len
        ultimately
        have "correct_frame G hp (ST',LT') maxl ins ?f'"
          by (unfold correct_frame_def) (auto simp add: sup_state_conv
              approx_stk_def approx_val_def split: err.split elim: conf_widen)
      }
      ultimately
      show ?thesis by (rule that)
    qed (insert xp', auto) — the other instructions don't generate exceptions

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

4.21.3 Single Instructions

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

```
lemmas [iff] = not_Err_eq

lemma Load_correct:
"⟦ wf_prog wt G;
    method (G,C) sig = Some (C,rT,maxs,maxl,ins,et);
    ins!pc = Load idx;
    wt_instr (ins!pc) G rT (phi C sig) maxs (length ins) et pc;
    Some state' = exec (G, None, hp, (stk,loc,C,sig,pc)#frs);
    G,phi ⊢JVM (None, hp, (stk,loc,C,sig,pc)#frs)√ ⟧
⟹ G,phi ⊢JVM state'√"
apply (clarsimp simp add: defs1)
apply (blast intro: approx_loc_imp_approx_val_sup)
done

lemma Store_correct:
"⟦ wf_prog wt G;
  method (G,C) sig = Some (C,rT,maxs,maxl,ins,et);
  ins!pc = Store idx;
  wt_instr (ins!pc) G rT (phi C sig) maxs (length ins) et pc;
  Some state' = exec (G, None, hp, (stk,loc,C,sig,pc)#frs);
  G,phi ⊢JVM (None, hp, (stk,loc,C,sig,pc)#frs)√ ⟧
⟹ G,phi ⊢JVM state'√"
apply (clarsimp simp add: defs1)
apply (blast intro: approx_loc_subst)
done


lemma LitPush_correct:
"⟦ wf_prog wt G;
    method (G,C) sig = Some (C,rT,maxs,maxl,ins,et);
```

```
    ins!pc = LitPush v;
    wt_instr (ins!pc) G rT (phi C sig) maxs (length ins) et pc;
    Some state' = exec (G, None, hp, (stk,loc,C,sig,pc)#frs);
    G,phi ⊢JVM (None, hp, (stk,loc,C,sig,pc)#frs)√ ⟧
⟹ G,phi ⊢JVM state'√"
apply (clarsimp simp add: defs1 sup_PTS_eq)
apply (blast dest: conf_litval intro: conf_widen)
done


lemma Cast_conf2:
  "⟦ wf_prog ok G; G,h⊢v::≼RefT rt; cast_ok G C h v;
      G⊢Class C≼T; is_class G C⟧
  ⟹ G,h⊢v::≼T"
apply (unfold cast_ok_def)
apply (frule widen_Class)
apply (elim exE disjE)
 apply (simp add: null)
apply (clarsimp simp add: conf_def obj_ty_def)
apply (cases v)
apply auto
done

lemmas defs2 = defs1 raise_system_xcpt_def

lemma Checkcast_correct:
"⟦ wt_jvm_prog G phi;
    method (G,C) sig = Some (C,rT,maxs,maxl,ins,et);
    ins!pc = Checkcast D;
    wt_instr (ins!pc) G rT (phi C sig) maxs (length ins) et pc;
    Some state' = exec (G, None, hp, (stk,loc,C,sig,pc)#frs) ;
    G,phi ⊢JVM (None, hp, (stk,loc,C,sig,pc)#frs)√;
    fst (exec_instr (ins!pc) G hp stk loc C sig pc frs) = None ⟧
⟹ G,phi ⊢JVM state'√"
apply (clarsimp simp add: defs2 wt_jvm_prog_def split: split_if_asm)
apply (blast intro: Cast_conf2)
done


lemma Getfield_correct:
"⟦ wt_jvm_prog G phi;
  method (G,C) sig = Some (C,rT,maxs,maxl,ins,et);
  ins!pc = Getfield F D;
  wt_instr (ins!pc) G rT (phi C sig) maxs (length ins) et pc;
  Some state' = exec (G, None, hp, (stk,loc,C,sig,pc)#frs) ;
  G,phi ⊢JVM (None, hp, (stk,loc,C,sig,pc)#frs)√;
  fst (exec_instr (ins!pc) G hp stk loc C sig pc frs) = None ⟧
⟹ G,phi ⊢JVM state'√"
apply (clarsimp simp add: defs2 wt_jvm_prog_def split_beta
                split: option.split split_if_asm)
apply (frule conf_widen)
apply assumption+
apply (drule conf_RefTD)
apply (clarsimp simp add: defs2)
```

```
apply (rule conjI)
 apply (drule widen_cfs_fields)
 apply assumption+
 apply (erule wf_prog_ws_prog)
 apply (erule conf_widen)
 prefer 2
  apply assumption
 apply (simp add: hconf_def oconf_def lconf_def)
 apply (elim allE)
 apply (erule impE, assumption)
 apply simp
 apply (elim allE)
 apply (erule impE, assumption)
 apply clarsimp
apply blast
done


lemma Putfield_correct:
"⟦ wf_prog wt G;
  method (G,C) sig = Some (C,rT,maxs,maxl,ins,et);
  ins!pc = Putfield F D;
  wt_instr (ins!pc) G rT (phi C sig) maxs (length ins) et pc;
  Some state' = exec (G, None, hp, (stk,loc,C,sig,pc)#frs) ;
  G,phi ⊢JVM (None, hp, (stk,loc,C,sig,pc)#frs)√;
  fst (exec_instr (ins!pc) G hp stk loc C sig pc frs) = None ⟧
⟹ G,phi ⊢JVM state'√"
apply (clarsimp simp add: defs2 split_beta split: option.split List.split split_if_asm)
apply (frule conf_widen)
prefer 2
  apply assumption
 apply assumption
apply (drule conf_RefTD)
apply clarsimp
apply (blast
       intro:
         hext_upd_obj approx_stk_sup_heap
         approx_loc_sup_heap
         hconf_field_update
         preallocated_field_update
         correct_frames_field_update conf_widen
       dest:
         widen_cfs_fields)
done


lemma New_correct:
"⟦ wf_prog wt G;
  method (G,C) sig = Some (C,rT,maxs,maxl,ins,et);
  ins!pc = New X;
  wt_instr (ins!pc) G rT (phi C sig) maxs (length ins) et pc;
  Some state' = exec (G, None, hp, (stk,loc,C,sig,pc)#frs) ;
  G,phi ⊢JVM (None, hp, (stk,loc,C,sig,pc)#frs)√;
  fst (exec_instr (ins!pc) G hp stk loc C sig pc frs) = None ⟧
```

```
⟹ G,phi ⊢JVM state'√"
proof -
  assume wf:   "wf_prog wt G"
  assume meth: "method (G,C) sig = Some (C,rT,maxs,maxl,ins,et)"
  assume ins:  "ins!pc = New X"
  assume wt:   "wt_instr (ins!pc) G rT (phi C sig) maxs (length ins) et pc"
  assume exec: "Some state' = exec (G, None, hp, (stk,loc,C,sig,pc)#frs)"
  assume conf: "G,phi ⊢JVM (None, hp, (stk,loc,C,sig,pc)#frs)√"
  assume no_x: "fst (exec_instr (ins!pc) G hp stk loc C sig pc frs) = None"

  from ins conf meth
  obtain ST LT where
    heap_ok:    "G⊢h hp√" and
    prealloc:   "preallocated hp" and
    phi_pc:     "phi C sig!pc = Some (ST,LT)" and
    is_class_C: "is_class G C" and
    frame:      "correct_frame G hp (ST,LT) maxl ins (stk, loc, C, sig, pc)" and
    frames:     "correct_frames G hp phi rT sig frs"
    by (auto simp add: correct_state_def iff del: not_None_eq)

  from phi_pc ins wt
  obtain ST' LT' where
    is_class_X: "is_class G X" and
    maxs:       "length ST < maxs" and
    suc_pc:     "Suc pc < length ins" and
    phi_suc:    "phi C sig ! Suc pc = Some (ST', LT')" and
    less:       "G ⊢ (Class X # ST, LT) <=s (ST', LT')"
    by (unfold wt_instr_def eff_def norm_eff_def) auto

  obtain oref xp' where
    new_Addr: "new_Addr hp = (oref,xp')"
    by (cases "new_Addr hp")
  with ins no_x
  obtain hp: "hp oref = None" and "xp' = None"
    by (auto dest: new_AddrD simp add: raise_system_xcpt_def)

  with exec ins meth new_Addr
  have state':
    "state' = Norm (hp(oref↦(X, init_vars (fields (G, X)))),
              (Addr oref # stk, loc, C, sig, Suc pc) # frs)"
    (is "state' = Norm (?hp', ?f # frs)")
    by simp
  moreover
  from wf hp heap_ok is_class_X
  have hp': "G ⊢h ?hp' √"
    by - (rule hconf_newref,
          auto simp add: oconf_def dest: fields_is_type)
  moreover
  from hp
  have sup: "hp ≤| ?hp'" by (rule hext_new)
  from hp frame less suc_pc wf
  have "correct_frame G ?hp' (ST', LT') maxl ins ?f"
    apply (unfold correct_frame_def sup_state_conv)
    apply (clarsimp simp add: conf_def fun_upd_apply approx_val_def)
```

```
    apply (blast intro: approx_stk_sup_heap approx_loc_sup_heap sup)
    done
  moreover
  from hp frames wf heap_ok is_class_X
  have "correct_frames G ?hp' phi rT sig frs"
    by - (rule correct_frames_newref,
          auto simp add: oconf_def dest: fields_is_type)
  moreover
  from hp prealloc have "preallocated ?hp'" by (rule preallocated_newref)
  ultimately
  show ?thesis
    by (simp add: is_class_C meth phi_suc correct_state_def del: not_None_eq)
qed

lemmas [simp del] = split_paired_Ex


lemma Invoke_correct:
"⟦ wt_jvm_prog G phi;
  method (G,C) sig = Some (C,rT,maxs,maxl,ins,et);
  ins ! pc = Invoke C' mn pTs;
  wt_instr (ins!pc) G rT (phi C sig) maxs (length ins) et pc;
  Some state' = exec (G, None, hp, (stk,loc,C,sig,pc)#frs) ;
  G,phi ⊢JVM (None, hp, (stk,loc,C,sig,pc)#frs)√;
  fst (exec_instr (ins!pc) G hp stk loc C sig pc frs) = None ⟧
⟹ G,phi ⊢JVM state'√"
proof -
  assume wtprog: "wt_jvm_prog G phi"
  assume method: "method (G,C) sig = Some (C,rT,maxs,maxl,ins,et)"
  assume ins:    "ins ! pc = Invoke C' mn pTs"
  assume wti:    "wt_instr (ins!pc) G rT (phi C sig) maxs (length ins) et pc"
  assume state': "Some state' = exec (G, None, hp, (stk,loc,C,sig,pc)#frs)"
  assume approx: "G,phi ⊢JVM (None, hp, (stk,loc,C,sig,pc)#frs)√"
  assume no_xcp: "fst (exec_instr (ins!pc) G hp stk loc C sig pc frs) = None"

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

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

  from ins wti phi_pc
  obtain apTs X ST LT D' rT body where
    is_class: "is_class G C'" and
    s:  "s = (rev apTs @ X # ST, LT)" and
```

```
    l:  "length apTs = length pTs" and
    X:  "G⊢ X ≼ Class C'" and
    w:  "∀(x, y)∈set (zip apTs pTs). G ⊢ x ≼ y" and
    mC':"method (G, C') (mn, pTs) = Some (D', rT, body)" and
    pc: "Suc pc < length ins" and
    eff: "G ⊢ norm_eff (Invoke C' mn pTs) G (Some s) <=' phi C sig!Suc pc"
    by (simp add: wt_instr_def eff_def del: not_None_eq)
       (elim exE conjE, rule that)

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

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

  from s ins frame
  obtain
    a_stk: "approx_stk G hp stk (rev apTs @ X # ST)" and
    a_loc: "approx_loc G hp loc LT" and
    suc_l: "length loc = Suc (length (snd sig) + maxl)"
    by (simp add: correct_frame_def)

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

  from oX X_Ref
  have oX_conf: "G,hp ⊢ oX ::≼ RefT T"
    by (simp add: approx_val_def)

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

  with state' method ins no_xcp oX_conf
  obtain ref where oX_Addr: "oX = Addr ref"
    by (auto simp add: raise_system_xcpt_def dest: conf_RefTD)

  with oX_conf X_Ref
  obtain obj D where
    loc:    "hp ref = Some obj" and
    obj_ty: "obj_ty obj = Class D" and
    D:      "G ⊢ Class D ≼ X"
    by (auto simp add: conf_def) blast
```

```
with X_Ref obtain X' where X': "X = Class X'"
  by (blast dest: widen_Class)

with X have X'_subcls: "G ⊢ X' ≼C C'"  by simp

with mC' wfprog
obtain D0 rT0 maxs0 maxl0 ins0 et0 where
  mX: "method (G, X') (mn, pTs) = Some (D0, rT0, maxs0, maxl0, ins0, et0)" "G⊢rT0≼rT"
  by (auto dest: subtype_widen_methd intro: that)

from X' D have D_subcls: "G ⊢ D ≼C X'" by simp

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

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

from is_class X'_subcls D_subcls
have is_class_D: "is_class G D" by (auto dest: subcls_is_class2)

with mD wfprog
obtain mD'':
  "method (G, D'') (mn, pTs) = Some (D'', rT', mxs', mxl', ins', et')"
  "is_class G D''"
  by (auto dest: wf_prog_ws_prog [THEN method_in_md])

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

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

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

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

have c_f: "correct_frame G hp ([], LT0) mxl' ins' ?f"
proof -
  from start LT0
  have sup_loc:
    "G ⊢ (OK (Class D'') # map OK pTs @ replicate mxl' Err) <=l LT0"
    (is "G ⊢ ?LT <=l LT0")
    by (simp add: wt_start_def sup_state_conv)
```

```
    have r: "approx_loc G hp (replicate mxl' undefined) (replicate mxl' Err)"
      by (simp add: approx_loc_def list_all2_def set_replicate_conv_if)

    from wfprog mD is_class_D
    have "G ⊢ Class D ≼ Class D''"
      by (auto dest: method_wf_mdecl)
    with obj_ty loc
    have a: "approx_val G hp (Addr ref) (OK (Class D''))"
      by (simp add: approx_val_def conf_def)

    from opTs w l l_o wfprog
    have "approx_stk G hp opTs (rev pTs)"
      by (auto elim!: approx_stk_all_widen simp add: zip_rev)
    hence "approx_stk G hp (rev opTs) pTs" by (subst approx_stk_rev)

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

    from this sup_loc wfprog
    have "approx_loc G hp ?lt LT0" by (rule approx_loc_widen)
    with start l_o l
    show ?thesis by (simp add: correct_frame_def)
  qed

  from state'_val heap_ok mD'' ins method phi_pc s X' l mX
       frames s' LT0 c_f is_class_C stk' oX_Addr frame prealloc and l
  show ?thesis
    apply (simp add: correct_state_def)
    apply (intro exI conjI)
       apply blast
      apply (rule l)
     apply (rule mX)
    apply (rule mD)
    done
qed

lemmas [simp del] = map_append

lemma Return_correct:
"⟦ wt_jvm_prog G phi;
  method (G,C) sig = Some (C,rT,maxs,maxl,ins,et);
  ins ! pc = Return;
  wt_instr (ins!pc) G rT (phi C sig) maxs (length ins) et pc;
  Some state' = exec (G, None, hp, (stk,loc,C,sig,pc)#frs) ;
  G,phi ⊢JVM (None, hp, (stk,loc,C,sig,pc)#frs)√ ⟧
⟹ G,phi ⊢JVM state'√"
proof -
  assume wt_prog: "wt_jvm_prog G phi"
  assume meth: "method (G,C) sig = Some (C,rT,maxs,maxl,ins,et)"
  assume ins: "ins ! pc = Return"
  assume wt: "wt_instr (ins!pc) G rT (phi C sig) maxs (length ins) et pc"
```

```
assume s': "Some state' = exec (G, None, hp, (stk,loc,C,sig,pc)#frs)"
assume correct: "G,phi ⊢JVM (None, hp, (stk,loc,C,sig,pc)#frs)√"

from wt_prog
obtain wfmb where wf: "wf_prog wfmb G" by (simp add: wt_jvm_prog_def)

from meth ins s'
have "frs = [] ⟹ ?thesis" by (simp add: correct_state_def)
moreover
{ fix f frs'
  assume frs': "frs = f#frs'"
  moreover
  obtain stk' loc' C' sig' pc' where
    f: "f = (stk',loc',C',sig',pc')" by (cases f)
  moreover
  obtain mn pt where
    sig: "sig = (mn,pt)" by (cases sig)
  moreover
  note meth ins s'
  ultimately
  have state':
    "state' = (None,hp,(hd stk#(drop (1+length pt) stk'),loc',C',sig',pc'+1)#frs')"
    (is "state' = (None,hp,?f'#frs')")
    by simp

  from correct meth
  obtain ST LT where
    hp_ok:  "G ⊢h hp √" and
    alloc:  "preallocated hp" and
    phi_pc: "phi C sig ! pc = Some (ST, LT)" and
    frame:  "correct_frame G hp (ST, LT) maxl ins (stk,loc,C,sig,pc)" and
    frames: "correct_frames G hp phi rT sig frs"
    by (simp add: correct_state_def, clarify, blast)

  from phi_pc ins wt
  obtain T ST' where "ST = T # ST'" "G ⊢ T ≼ rT"
    by (simp add: wt_instr_def) blast
  with wf frame
  have hd_stk: "G,hp ⊢ (hd stk) ::≼ rT"
    by (auto simp add: correct_frame_def elim: conf_widen)

  from f frs' frames sig
  obtain apTs ST0' ST' LT' D D' D'' rT' rT'' maxs' maxl' ins' et' body where
    phi':   "phi C' sig' ! pc' = Some (ST',LT')" and
    class': "is_class G C'" and
    meth':  "method (G,C') sig' = Some (C',rT',maxs',maxl',ins',et')" and
    ins':   "ins' ! pc' = Invoke D' mn pt" and
    frame': "correct_frame G hp (ST', LT') maxl' ins' f" and
    frames':"correct_frames G hp phi rT' sig' frs'" and
    rT'':   "G ⊢ rT ≼ rT''" and
    meth'': "method (G, D) sig = Some (D'', rT'', body)" and
    ST0':   "ST' = rev apTs @ Class D # ST0'" and
    len':   "length apTs = length pt"
    by clarsimp blast
```

```
    from f frame'
    obtain
      stk': "approx_stk G hp stk' ST'" and
      loc': "approx_loc G hp loc' LT'" and
      pc':  "pc' < length ins'" and
      lloc':"length loc' = Suc (length (snd sig') + maxl')"
      by (simp add: correct_frame_def)

    from wt_prog class' meth' pc'
    have "wt_instr (ins'!pc') G rT' (phi C' sig') maxs' (length ins') et' pc'"
      by (rule wt_jvm_prog_impl_wt_instr)
    with ins' phi' sig
    obtain apTs ST0 X ST'' LT'' body' rT0 mD where
      phi_suc: "phi C' sig' ! Suc pc' = Some (ST'', LT'')" and
      ST0:     "ST' = rev apTs @ X # ST0" and
      len:     "length apTs = length pt" and
      less:    "G ⊢ (rT0 # ST0, LT') <=s (ST'', LT'')" and
      methD':  "method (G, D') sig = Some (mD, rT0, body')" and
      lessD':  "G ⊢ X ≼ Class D'" and
      suc_pc': "Suc pc' < length ins'"
      by (clarsimp simp add: wt_instr_def eff_def norm_eff_def) blast

    from len len' ST0 ST0'
    have "X = Class D" by simp
    with lessD'
    have "G ⊢ D ≼C D'" by simp
    moreover
    note wf meth'' methD'
    ultimately
    have "G ⊢ rT'' ≼ rT0" by (auto dest: subcls_widen_methd)
    with wf hd_stk rT''
    have hd_stk': "G,hp ⊢ (hd stk) ::≼ rT0" by (auto elim: conf_widen widen_trans)

    have frame'':
      "correct_frame G hp (ST'',LT'') maxl' ins' ?f'"
    proof -
      from wf hd_stk' len stk' less ST0
      have "approx_stk G hp (hd stk # drop (1+length pt) stk') ST''"
        by (auto simp add: sup_state_conv
                 dest!: approx_stk_append elim: conf_widen)
      moreover
      from wf loc' less
      have "approx_loc G hp loc' LT''" by (simp add: sup_state_conv) blast
      moreover
      note suc_pc' lloc'
      ultimately
      show ?thesis by (simp add: correct_frame_def)
    qed

    with state' frs' f meth hp_ok hd_stk phi_suc frames' meth' phi' class' alloc
    have ?thesis by (simp add: correct_state_def)
  }
  ultimately
```

```
  show ?thesis by (cases frs) blast+
qed

lemmas [simp] = map_append

lemma Goto_correct:
"⟦ wf_prog wt G;
  method (G,C) sig = Some (C,rT,maxs,maxl,ins,et);
  ins ! pc = Goto branch;
  wt_instr (ins!pc) G rT (phi C sig) maxs (length ins) et pc;
  Some state' = exec (G, None, hp, (stk,loc,C,sig,pc)#frs) ;
  G,phi ⊢JVM (None, hp, (stk,loc,C,sig,pc)#frs)√ ⟧
⟹ G,phi ⊢JVM state'√"
apply (clarsimp simp add: defs2)
apply fast
done


lemma Ifcmpeq_correct:
"⟦ wf_prog wt G;
  method (G,C) sig = Some (C,rT,maxs,maxl,ins,et);
  ins ! pc = Ifcmpeq branch;
  wt_instr (ins!pc) G rT (phi C sig) maxs (length ins) et pc;
  Some state' = exec (G, None, hp, (stk,loc,C,sig,pc)#frs) ;
  G,phi ⊢JVM (None, hp, (stk,loc,C,sig,pc)#frs)√ ⟧
⟹ G,phi ⊢JVM state'√"
apply (clarsimp simp add: defs2)
apply fast
done

lemma Pop_correct:
"⟦ wf_prog wt G;
  method (G,C) sig = Some (C,rT,maxs,maxl,ins,et);
  ins ! pc = Pop;
  wt_instr (ins!pc) G rT (phi C sig) maxs (length ins) et pc;
  Some state' = exec (G, None, hp, (stk,loc,C,sig,pc)#frs) ;
  G,phi ⊢JVM (None, hp, (stk,loc,C,sig,pc)#frs)√ ⟧
⟹ G,phi ⊢JVM state'√"
apply (clarsimp simp add: defs2)
apply fast
done

lemma Dup_correct:
"⟦ wf_prog wt G;
  method (G,C) sig = Some (C,rT,maxs,maxl,ins,et);
  ins ! pc = Dup;
  wt_instr (ins!pc) G rT (phi C sig) maxs (length ins) et pc;
  Some state' = exec (G, None, hp, (stk,loc,C,sig,pc)#frs) ;
  G,phi ⊢JVM (None, hp, (stk,loc,C,sig,pc)#frs)√ ⟧
⟹ G,phi ⊢JVM state'√"
apply (clarsimp simp add: defs2)
apply (blast intro: conf_widen)
done
```

lemma *Dup_x1_correct:*
```
"⟦ wf_prog wt G;
  method (G,C) sig = Some (C,rT,maxs,maxl,ins,et);
  ins ! pc = Dup_x1;
  wt_instr (ins!pc) G rT (phi C sig) maxs (length ins) et pc;
  Some state' = exec (G, None, hp, (stk,loc,C,sig,pc)#frs) ;
  G,phi ⊢JVM (None, hp, (stk,loc,C,sig,pc)#frs)√ ⟧
⟹ G,phi ⊢JVM state'√"
```
apply *(clarsimp simp add: defs2)*
apply *(blast intro: conf_widen)*
done

lemma *Dup_x2_correct:*
```
"⟦ wf_prog wt G;
  method (G,C) sig = Some (C,rT,maxs,maxl,ins,et);
  ins ! pc = Dup_x2;
  wt_instr (ins!pc) G rT (phi C sig) maxs (length ins) et pc;
  Some state' = exec (G, None, hp, (stk,loc,C,sig,pc)#frs) ;
  G,phi ⊢JVM (None, hp, (stk,loc,C,sig,pc)#frs)√ ⟧
⟹ G,phi ⊢JVM state'√"
```
apply *(clarsimp simp add: defs2)*
apply *(blast intro: conf_widen)*
done

lemma *Swap_correct:*
```
"⟦ wf_prog wt G;
  method (G,C) sig = Some (C,rT,maxs,maxl,ins,et);
  ins ! pc = Swap;
  wt_instr (ins!pc) G rT (phi C sig) maxs (length ins) et pc;
  Some state' = exec (G, None, hp, (stk,loc,C,sig,pc)#frs) ;
  G,phi ⊢JVM (None, hp, (stk,loc,C,sig,pc)#frs)√ ⟧
⟹ G,phi ⊢JVM state'√"
```
apply *(clarsimp simp add: defs2)*
apply *(blast intro: conf_widen)*
done

lemma *IAdd_correct:*
```
"⟦ wf_prog wt G;
  method (G,C) sig = Some (C,rT,maxs,maxl,ins,et);
  ins ! pc = IAdd;
  wt_instr (ins!pc) G rT (phi C sig) maxs (length ins) et pc;
  Some state' = exec (G, None, hp, (stk,loc,C,sig,pc)#frs) ;
  G,phi ⊢JVM (None, hp, (stk,loc,C,sig,pc)#frs)√ ⟧
⟹ G,phi ⊢JVM state'√"
```
apply *(clarsimp simp add: defs2 approx_val_def conf_def)*
apply *blast*
done

lemma *Throw_correct:*
```
"⟦ wf_prog wt G;
  method (G,C) sig = Some (C,rT,maxs,maxl,ins,et);
  ins ! pc = Throw;
  Some state' = exec (G, None, hp, (stk,loc,C,sig,pc)#frs) ;
  G,phi ⊢JVM (None, hp, (stk,loc,C,sig,pc)#frs)√;
```

```
  fst (exec_instr (ins!pc) G hp stk loc C sig pc frs) = None ⟧
⟹ G,phi ⊢JVM state'√"
  by simp
```

The next theorem collects the results of the sections above, i.e. exception handling and the execution step for each instruction. It states type safety for single step execution: in well-typed programs, a conforming state is transformed into another conforming state when one instruction is executed.

```
theorem instr_correct:
"⟦ wt_jvm_prog G phi;
  method (G,C) sig = Some (C,rT,maxs,maxl,ins,et);
  Some state' = exec (G, None, hp, (stk,loc,C,sig,pc)#frs);
  G,phi ⊢JVM (None, hp, (stk,loc,C,sig,pc)#frs)√ ⟧
⟹ G,phi ⊢JVM state'√"
apply (frule wt_jvm_prog_impl_wt_instr_cor)
apply assumption+
apply (cases "fst (exec_instr (ins!pc) G hp stk loc C sig pc frs)")
defer
apply (erule xcpt_correct, assumption+)
apply (cases "ins!pc")
prefer 8
apply (rule Invoke_correct, assumption+)
prefer 8
apply (rule Return_correct, assumption+)
prefer 5
apply (rule Getfield_correct, assumption+)
prefer 6
apply (rule Checkcast_correct, assumption+)

apply (unfold wt_jvm_prog_def)
apply (rule Load_correct, assumption+)
apply (rule Store_correct, assumption+)
apply (rule LitPush_correct, assumption+)
apply (rule New_correct, assumption+)
apply (rule Putfield_correct, assumption+)
apply (rule Pop_correct, assumption+)
apply (rule Dup_correct, assumption+)
apply (rule Dup_x1_correct, assumption+)
apply (rule Dup_x2_correct, assumption+)
apply (rule Swap_correct, assumption+)
apply (rule IAdd_correct, assumption+)
apply (rule Goto_correct, assumption+)
apply (rule Ifcmpeq_correct, assumption+)
apply (rule Throw_correct, assumption+)
done
```

4.21.4 Main

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

```
lemma BV_correct_1 [rule_format]:
"⋀state. ⟦ wt_jvm_prog G phi; G,phi ⊢JVM state√⟧
 ⟹ exec (G,state) = Some state' ⟶ G,phi ⊢JVM state'√"
apply (simp only: split_tupled_all)
apply (rename_tac xp hp frs)
apply (case_tac xp)
 apply (case_tac frs)
  apply simp
 apply (simp only: split_tupled_all)
 apply hypsubst
 apply (frule correct_state_impl_Some_method)
 apply (force intro: instr_correct)
apply (case_tac frs)
apply simp_all
done


lemma L0:
  "⟦ xp=None; frs≠[] ⟧ ⟹ (∃state'. exec (G,xp,hp,frs) = Some state')"
by (clarsimp simp add: neq_Nil_conv split_beta)

lemma L1:
  "⟦wt_jvm_prog G phi; G,phi ⊢JVM (xp,hp,frs)√; xp=None; frs≠[]⟧
  ⟹ ∃state'. exec(G,xp,hp,frs) = Some state' ∧ G,phi ⊢JVM state'√"
apply (drule L0)
apply assumption
apply (fast intro: BV_correct_1)
done

theorem BV_correct [rule_format]:
"⟦ wt_jvm_prog G phi; G ⊢ s -jvm→ t ⟧ ⟹ G,phi ⊢JVM s√ ⟶ G,phi ⊢JVM t√"
apply (unfold exec_all_def)
apply (erule rtrancl_induct)
 apply simp
apply (auto intro: BV_correct_1)
done


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

lemma
  fixes G :: jvm_prog ("Γ")
  assumes wf: "wf_prog wf_mb Γ"
  shows hconf_start: "Γ ⊢h (start_heap Γ) √"
```

```
  apply (unfold hconf_def start_heap_def)
  apply (auto simp add: fun_upd_apply blank_def oconf_def split: split_if_asm)
  apply (simp add: fields_is_type
          [OF _ wf [THEN wf_prog_ws_prog]
                is_class_xcpt [OF wf [THEN wf_prog_ws_prog]]])+
  done

lemma
  fixes G :: jvm_prog ("Γ") and Phi :: prog_type ("Φ")
  shows BV_correct_initial:
  "wt_jvm_prog Γ Φ ⟹ is_class Γ C ⟹ method (Γ,C) (m,[]) = Some (C, b)
  ⟹ Γ,Φ ⊢JVM start_state G C m √"
  apply (cases b)
  apply (unfold  start_state_def)
  apply (unfold correct_state_def)
  apply (auto simp add: preallocated_start)
   apply (simp add: wt_jvm_prog_def hconf_start)
  apply (drule wt_jvm_prog_impl_wt_start, assumption+)
  apply (clarsimp simp add: wt_start_def)
  apply (auto simp add: correct_frame_def)
   apply (simp add: approx_stk_def sup_state_conv)
  apply (auto simp add: sup_state_conv approx_val_def dest!: widen_RefT split: err.splits)
  done

theorem
  fixes G :: jvm_prog ("Γ") and Phi :: prog_type ("Φ")
  assumes welltyped:   "wt_jvm_prog Γ Φ" and
          main_method: "is_class Γ C" "method (Γ,C) (m,[]) = Some (C, b)"
  shows typesafe:
  "G ⊢ start_state Γ C m -jvm→ s  ⟹  Γ,Φ ⊢JVM s √"
proof -
  from welltyped main_method
  have "Γ,Φ ⊢JVM start_state Γ C m √" by (rule BV_correct_initial)
  moreover
  assume "G ⊢ start_state Γ C m -jvm→ s"
  ultimately
  show "Γ,Φ ⊢JVM s √" using welltyped by - (rule BV_correct)
qed

end
```

4.22 Welltyped Programs produce no Type Errors

```
theory BVNoTypeError
imports "../JVM/JVMDefensive" BVSpecTypeSafe
begin
```

Some simple lemmas about the type testing functions of the defensive JVM:

```
lemma typeof_NoneD [simp,dest]:
  "typeof (λv. None) v = Some x ⟹ ¬isAddr v"
  by (cases v) auto

lemma isRef_def2:
  "isRef v = (v = Null ∨ (∃loc. v = Addr loc))"
  by (cases v) (auto simp add: isRef_def)

lemma app'Store[simp]:
  "app' (Store idx, G, pc, maxs, rT, (ST,LT)) = (∃T ST'. ST = T#ST' ∧ idx < length LT)"
  by (cases ST, auto)

lemma app'GetField[simp]:
  "app' (Getfield F C, G, pc, maxs, rT, (ST,LT)) =
  (∃oT vT ST'. ST = oT#ST' ∧ is_class G C ∧
  field (G,C) F = Some (C,vT) ∧ G ⊢ oT ≼ Class C)"
  by (cases ST, auto)

lemma app'PutField[simp]:
"app' (Putfield F C, G,  pc, maxs, rT, (ST,LT)) =
 (∃vT vT' oT ST'. ST = vT#oT#ST' ∧ is_class G C ∧
  field (G,C) F = Some (C, vT') ∧
  G ⊢ oT ≼ Class C ∧ G ⊢ vT ≼ vT')"
  apply rule
  defer
  apply clarsimp
  apply (cases ST)
  apply simp
  apply (cases "tl ST")
  apply auto
  done

lemma app'Checkcast[simp]:
"app' (Checkcast C, G, pc, maxs, rT, (ST,LT)) =
 (∃rT ST'. ST = RefT rT#ST' ∧ is_class G C)"
apply rule
defer
apply clarsimp
apply (cases ST)
apply simp
apply (cases "hd ST")
defer
apply simp
apply simp
done
```

```
lemma app'Pop[simp]:
  "app' (Pop, G, pc, maxs, rT, (ST,LT)) = (∃T ST'. ST = T#ST')"
  by (cases ST, auto)


lemma app'Dup[simp]:
  "app' (Dup, G, pc, maxs, rT, (ST,LT)) =
  (∃T ST'. ST = T#ST' ∧ length ST < maxs)"
  by (cases ST, auto)


lemma app'Dup_x1[simp]:
  "app' (Dup_x1, G, pc, maxs, rT, (ST,LT)) =
  (∃T1 T2 ST'. ST = T1#T2#ST' ∧ length ST < maxs)"
  by (cases ST, simp, cases "tl ST", auto)


lemma app'Dup_x2[simp]:
  "app' (Dup_x2, G, pc, maxs, rT, (ST,LT)) =
  (∃T1 T2 T3 ST'. ST = T1#T2#T3#ST' ∧ length ST < maxs)"
  by (cases ST, simp, cases "tl ST", simp, cases "tl (tl ST)", auto)


lemma app'Swap[simp]:
  "app' (Swap, G, pc, maxs, rT, (ST,LT)) = (∃T1 T2 ST'. ST = T1#T2#ST')"
  by (cases ST, simp, cases "tl ST", auto)


lemma app'IAdd[simp]:
  "app' (IAdd, G, pc, maxs, rT, (ST,LT)) =
  (∃ST'. ST = PrimT Integer#PrimT Integer#ST')"
  apply (cases ST)
  apply simp
  apply simp
  apply (case_tac a)
  apply auto
  apply (case_tac prim_ty)
  apply auto
  apply (case_tac prim_ty)
  apply auto
  apply (case_tac list)
  apply auto
  apply (case_tac a)
  apply auto
  apply (case_tac prim_ty)
  apply auto
  done


lemma app'Ifcmpeq[simp]:
  "app' (Ifcmpeq b, G, pc, maxs, rT, (ST,LT)) =
  (∃T1 T2 ST'. ST = T1#T2#ST' ∧ 0 ≤ b + int pc ∧
  ((∃p. T1 = PrimT p ∧ T1 = T2) ∨
  (∃r r'. T1 = RefT r ∧ T2 = RefT r')))"
```

```
  apply auto
  apply (cases ST)
  apply simp
  apply (cases "tl ST")
  apply (case_tac a)
  apply auto
  done


lemma app'Return[simp]:
  "app' (Return, G, pc, maxs, rT, (ST,LT)) =
  (∃T ST'. ST = T#ST'∧ G ⊢ T ≼ rT)"
  by (cases ST, auto)


lemma app'Throw[simp]:
  "app' (Throw, G, pc, maxs, rT, (ST,LT)) =
  (∃ST' r. ST = RefT r#ST')"
  apply (cases ST, simp)
  apply (cases "hd ST")
  apply auto
  done


lemma app'Invoke[simp]:
"app' (Invoke C mn fpTs, G, pc, maxs, rT, ST, LT) =
 (∃apTs X ST' mD' rT' b'.
  ST = (rev apTs) @ X # ST' ∧
  length apTs = length fpTs ∧ is_class G C ∧
  (∀(aT,fT)∈set(zip apTs fpTs). G ⊢ aT ≼ fT) ∧
  method (G,C) (mn,fpTs) = Some (mD', rT', b') ∧ G ⊢ X ≼ Class C)"
  (is "?app ST LT = ?P ST LT")
proof
  assume "?P ST LT" thus "?app ST LT" by (auto simp add: list_all2_def)
next
  assume app: "?app ST LT"
  hence l: "length fpTs < length ST" by simp
  obtain xs ys where xs: "ST = xs @ ys" "length xs = length fpTs"
  proof -
    have "ST = take (length fpTs) ST @ drop (length fpTs) ST" by simp
    moreover from l have "length (take (length fpTs) ST) = length fpTs"
      by simp
    ultimately show ?thesis ..
  qed
  obtain apTs where
    "ST = (rev apTs) @ ys" and "length apTs = length fpTs"
  proof -
    from xs(1) have "ST = rev (rev xs) @ ys" by simp
    then show thesis by (rule that) (simp add: xs(2))
  qed
  moreover
  from l xs obtain X ST' where "ys = X#ST'" by (auto simp add: neq_Nil_conv)
  ultimately
  have "ST = (rev apTs) @ X # ST'" "length apTs = length fpTs" by auto
```

```
  with app
  show "?P ST LT"
    apply (clarsimp simp add: list_all2_def)
    apply ((rule exI)+, (rule conjI)?)+
    apply auto
    done
qed

lemma approx_loc_len [simp]:
  "approx_loc G hp loc LT ⟹ length loc = length LT"
  by (simp add: approx_loc_def list_all2_def)

lemma approx_stk_len [simp]:
  "approx_stk G hp stk ST ⟹ length stk = length ST"
  by (simp add: approx_stk_def)

lemma isRefI [intro, simp]: "G,hp ⊢ v ::≼ RefT T ⟹ isRef v"
  apply (drule conf_RefTD)
  apply (auto simp add: isRef_def)
  done

lemma isIntgI [intro, simp]: "G,hp ⊢ v ::≼ PrimT Integer ⟹ isIntg v"
  apply (unfold conf_def)
  apply auto
  apply (erule widen.cases)
  apply auto
  apply (cases v)
  apply auto
  done

lemma list_all2_approx:
  "⋀s. list_all2 (approx_val G hp) s (map OK S) =
       list_all2 (conf G hp) s S"
  apply (induct S)
  apply (auto simp add: list_all2_Cons2 approx_val_def)
  done

lemma list_all2_conf_widen:
  "wf_prog mb G ⟹
  list_all2 (conf G hp) a b ⟹
  list_all2 (λx y. G ⊢ x ≼ y) b c ⟹
  list_all2 (conf G hp) a c"
  apply (rule list_all2_trans)
  defer
  apply assumption
  apply assumption
  apply (drule conf_widen, assumption+)
  done
```

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

```
theorem no_type_error:
  assumes welltyped: "wt_jvm_prog G Phi" and conforms: "G,Phi ⊢JVM s √"
  shows "exec_d G (Normal s) ≠ TypeError"
```

```
proof -
  from welltyped obtain mb where wf: "wf_prog mb G" by (fast dest: wt_jvm_progD)

  obtain xcp hp frs where s [simp]: "s = (xcp, hp, frs)" by (cases s)

  from conforms have "xcp ≠ None ∨ frs = [] ⟹ check G s"
    by (unfold correct_state_def check_def) auto
  moreover {
    assume "¬(xcp ≠ None ∨ frs = [])"
    then obtain stk loc C sig pc frs' where
      xcp [simp]: "xcp = None" and
      frs [simp]: "frs = (stk,loc,C,sig,pc)#frs'"
      by (clarsimp simp add: neq_Nil_conv) fast

    from conforms obtain  ST LT rT maxs maxl ins et where
      hconf:  "G ⊢h hp √" and
      "class":  "is_class G C" and
      meth:   "method (G, C) sig = Some (C, rT, maxs, maxl, ins, et)" and
      phi:    "Phi C sig ! pc = Some (ST,LT)" and
      frame:  "correct_frame G hp (ST,LT) maxl ins (stk,loc,C,sig,pc)" and
      frames: "correct_frames G hp Phi rT sig frs'"
      by (auto simp add: correct_state_def) (rule that)

    from frame obtain
      stk: "approx_stk G hp stk ST" and
      loc: "approx_loc G hp loc LT" and
      pc:  "pc < length ins" and
      len: "length loc = length (snd sig) + maxl + 1"
      by (auto simp add: correct_frame_def)

    note approx_val_def [simp]

    from welltyped meth conforms
    have "wt_instr (ins!pc) G rT (Phi C sig) maxs (length ins) et pc"
      by simp (rule wt_jvm_prog_impl_wt_instr_cor)
    then obtain
      app': "app (ins!pc) G maxs rT pc et (Phi C sig!pc) " and
      eff: "∀(pc', s')∈set (eff (ins ! pc) G pc et (Phi C sig ! pc)). pc' < length ins"
      by (simp add: wt_instr_def phi) blast

    from eff
    have pc': "∀pc' ∈ set (succs (ins!pc) pc). pc' < length ins"
      by (simp add: eff_def) blast

    from app' phi
    have app:
      "xcpt_app (ins!pc) G pc et ∧ app' (ins!pc, G, pc, maxs, rT, (ST,LT))"
      by (clarsimp simp add: app_def)

    with eff stk loc pc'
    have "check_instr (ins!pc) G hp stk loc C sig pc maxs frs'"
    proof (cases "ins!pc")
      case (Getfield F C)
      with app stk loc phi obtain v vT stk' where
```

```
      "class": "is_class G C" and
      field: "field (G, C) F = Some (C, vT)" and
      stk:   "stk = v # stk'" and
      conf:  "G,hp ⊢ v ::≼ Class C"
      apply clarsimp
      apply (blast dest: conf_widen [OF wf])
      done
    from conf have isRef: "isRef v" ..
    moreover {
      assume "v ≠ Null"
      with conf field isRef wf
      have "∃D vs. hp (the_Addr v) = Some (D,vs) ∧ G ⊢ D ≼C C"
        by (auto dest!: non_np_objD)
    }
    ultimately show ?thesis using Getfield field "class" stk hconf wf
      apply clarsimp
      apply (fastsimp intro: wf_prog_ws_prog
        dest!: hconfD widen_cfs_fields oconf_objD)
      done
  next
    case (Putfield F C)
    with app stk loc phi obtain v ref vT stk' where
      "class": "is_class G C" and
      field: "field (G, C) F = Some (C, vT)" and
      stk:   "stk = v # ref # stk'" and
      confv: "G,hp ⊢ v ::≼ vT" and
      confr: "G,hp ⊢ ref ::≼ Class C"
      apply clarsimp
      apply (blast dest: conf_widen [OF wf])
      done
    from confr have isRef: "isRef ref" ..
    moreover {
      assume "ref ≠ Null"
      with confr field isRef wf
      have "∃D vs. hp (the_Addr ref) = Some (D,vs) ∧ G ⊢ D ≼C C"
        by (auto dest: non_np_objD)
    }
    ultimately show ?thesis using Putfield field "class" stk confv
      by clarsimp
  next
    case (Invoke C mn ps)
    with app
    obtain apTs X ST' where
      ST: "ST = rev apTs @ X # ST'" and
      ps: "length apTs = length ps" and
      w:   "∀(x, y)∈set (zip apTs ps). G ⊢ x ≼ y" and
      C:   "G ⊢ X ≼ Class C" and
      mth: "method (G, C) (mn, ps) ≠ None"
      by (simp del: app'.simps) blast

    from ST stk
    obtain aps x stk' where
      stk': "stk = aps @ x # stk'" and
      aps: "approx_stk G hp aps (rev apTs)" and
```

```
      x: "G,hp ⊢ x ::≼ X" and
      l: "length aps = length apTs"
      by (auto dest!: approx_stk_append)

    from stk' l ps
    have [simp]: "stk!length ps = x" by (simp add: nth_append)
    from stk' l ps
    have [simp]: "take (length ps) stk = aps" by simp
    from w ps
    have widen: "list_all2 (λx y. G ⊢ x ≼ y) apTs ps"
      by (simp add: list_all2_def)

    from stk' l ps have "length ps < length stk" by simp
    moreover
    from wf x C
    have x: "G,hp ⊢ x ::≼ Class C" by (rule conf_widen)
    hence "isRef x" by simp
    moreover
    { assume "x ≠ Null"
      with x
      obtain D fs where
        hp: "hp (the_Addr x) = Some (D,fs)" and
        D:  "G ⊢ D ≼C C"
        by - (drule non_npD, assumption, clarsimp, blast)
      hence "hp (the_Addr x) ≠ None" (is ?P1) by simp
      moreover
      from wf mth hp D
      have "method (G, cname_of hp x) (mn, ps) ≠ None" (is ?P2)
        by (auto dest: subcls_widen_methd)
      moreover
      from aps have "list_all2 (conf G hp) aps (rev apTs)"
        by (simp add: list_all2_approx approx_stk_def approx_loc_def)
      hence "list_all2 (conf G hp) (rev aps) (rev (rev apTs))"
        by (simp only: list_all2_rev)
      hence "list_all2 (conf G hp) (rev aps) apTs" by simp
      with wf widen
      have "list_all2 (conf G hp) (rev aps) ps" (is ?P3)
        by - (rule list_all2_conf_widen)
      ultimately
      have "?P1 ∧ ?P2 ∧ ?P3" by blast
    }
    moreover
    note Invoke
    ultimately
    show ?thesis by simp
  next
    case Return with stk app phi meth frames
    show ?thesis
      apply clarsimp
      apply (drule conf_widen [OF wf], assumption)
      apply (clarsimp simp add: neq_Nil_conv isRef_def2)
      done
  qed auto
  hence "check G s" by (simp add: check_def meth pc)
```

```
  } ultimately
  have "check G s" by blast
  thus "exec_d G (Normal s) ≠ TypeError" ..
qed
```

The theorem above tells us that, in welltyped programs, the defensive machine reaches the same result as the aggressive one (after arbitrarily many steps).

```
theorem welltyped_aggressive_imp_defensive:
  "wt_jvm_prog G Phi ⟹ G,Phi ⊢JVM s √ ⟹ G ⊢ s -jvm→ t
  ⟹ G ⊢ (Normal s) -jvmd→ (Normal t)"
  apply (unfold exec_all_def)
  apply (erule rtrancl_induct)
   apply (simp add: exec_all_d_def)
  apply simp
  apply (fold exec_all_def)
  apply (frule BV_correct, assumption+)
  apply (drule no_type_error, assumption, drule no_type_error_commutes, simp)
  apply (simp add: exec_all_d_def)
  apply (rule rtrancl_trans, assumption)
  apply blast
  done


lemma neq_TypeError_eq [simp]: "s ≠ TypeError = (∃s'. s = Normal s')"
  by (cases s, auto)

theorem no_type_errors:
  "wt_jvm_prog G Phi ⟹ G,Phi ⊢JVM s √
  ⟹ G ⊢ (Normal s) -jvmd→ t ⟹ t ≠ TypeError"
  apply (unfold exec_all_d_def)
  apply (erule rtrancl_induct)
   apply simp
  apply (fold exec_all_d_def)
  apply (auto dest: defensive_imp_aggressive BV_correct no_type_error)
  done

corollary no_type_errors_initial:
  fixes G ("Γ") and Phi ("Φ")
  assumes wt: "wt_jvm_prog Γ Φ"
  assumes is_class: "is_class Γ C"
    and method: "method (Γ,C) (m,[]) = Some (C, b)"
    and m: "m ≠ init"
  defines start: "s ≡ start_state Γ C m"

  assumes s: "Γ ⊢ (Normal s) -jvmd→ t"
  shows "t ≠ TypeError"
proof -
  from wt is_class method have "Γ,Φ ⊢JVM s √"
    unfolding start by (rule BV_correct_initial)
  from wt this s show ?thesis by (rule no_type_errors)
qed
```

As corollary we get that the aggressive and the defensive machine are equivalent for welltyped programs (if started in a conformant state or in the canonical start state)

```
corollary welltyped_commutes:
  fixes G ("Γ") and Phi ("Φ")
  assumes wt: "wt_jvm_prog Γ Φ" and *: "Γ,Φ ⊢JVM s √"
  shows "Γ ⊢ (Normal s) -jvmd→ (Normal t) = Γ ⊢ s -jvm→ t"
  apply rule
   apply (erule defensive_imp_aggressive, rule welltyped_aggressive_imp_defensive)
    apply (rule wt)
   apply (rule *)
  apply assumption
  done

corollary welltyped_initial_commutes:
  fixes G ("Γ") and Phi ("Φ")
  assumes wt: "wt_jvm_prog Γ Φ"
  assumes is_class: "is_class Γ C"
    and method: "method (Γ,C) (m,[]) = Some (C, b)"
    and m: "m ≠ init"
  defines start: "s ≡ start_state Γ C m"
  shows "Γ ⊢ (Normal s) -jvmd→ (Normal t) = Γ ⊢ s -jvm→ t"
proof -
  from wt is_class method have "Γ,Φ ⊢JVM s √"
    unfolding start by (rule BV_correct_initial)
  with wt show ?thesis by (rule welltyped_commutes)
qed

end
```

4.23 Kildall for the JVM

```
theory JVM
imports Typing_Framework_JVM
begin

definition kiljvm :: "jvm_prog ⇒ nat ⇒ nat ⇒ ty ⇒ exception_table ⇒
             instr list ⇒ JVMType.state list ⇒ JVMType.state list" where
  "kiljvm G maxs maxr rT et bs ==
  kildall (JVMType.le G maxs maxr) (JVMType.sup G maxs maxr) (exec G maxs rT et bs)"

definition wt_kil :: "jvm_prog ⇒ cname ⇒ ty list ⇒ ty ⇒ nat ⇒ nat ⇒
             exception_table ⇒ instr list ⇒ bool" where
  "wt_kil G C pTs rT mxs mxl et ins ==
   check_bounded ins et ∧ 0 < size ins ∧
   (let first  = Some ([],(OK (Class C))#((map OK pTs))@(replicate mxl Err));
        start  = OK first#(replicate (size ins - 1) (OK None));
        result = kiljvm G mxs (1+size pTs+mxl) rT et ins start
    in ∀n < size ins. result!n ≠ Err)"

definition wt_jvm_prog_kildall :: "jvm_prog ⇒ bool" where
  "wt_jvm_prog_kildall G ==
  wf_prog (λG C (sig,rT,(maxs,maxl,b,et)). wt_kil G C (snd sig) rT maxs maxl et b) G"

theorem is_bcv_kiljvm:
  "⟦ wf_prog wf_mb G; bounded (exec G maxs rT et bs) (size bs) ⟧ ⟹
      is_bcv (JVMType.le G maxs maxr) Err (exec G maxs rT et bs)
             (size bs) (states G maxs maxr) (kiljvm G maxs maxr rT et bs)"
  apply (unfold kiljvm_def sl_triple_conv)
  apply (rule is_bcv_kildall)
       apply (simp (no_asm) add: sl_triple_conv [symmetric])
       apply (force intro!: semilat_JVM_slI dest: wf_acyclic
         simp add: symmetric sl_triple_conv)
      apply (simp (no_asm) add: JVM_le_unfold)
      apply (blast intro!: order_widen wf_converse_subcls1_impl_acc_subtype
                   dest: wf_subcls1 wf_acyclic wf_prog_ws_prog)
     apply (simp add: JVM_le_unfold)
    apply (erule exec_pres_type)
   apply assumption
  apply (drule wf_prog_ws_prog, erule exec_mono, assumption)
  done

lemma subset_replicate: "set (replicate n x) ⊆ {x}"
  by (induct n) auto

lemma in_set_replicate:
  "x ∈ set (replicate n y) ⟹ x = y"
proof -
  assume "x ∈ set (replicate n y)"
  also have "set (replicate n y) ⊆ {y}" by (rule subset_replicate)
  finally have "x ∈ {y}" .
  thus ?thesis by simp
qed
```

```
theorem wt_kil_correct:
  assumes wf:  "wf_prog wf_mb G"
  assumes C:   "is_class G C"
  assumes pTs: "set pTs ⊆ types G"

  assumes wtk: "wt_kil G C pTs rT maxs mxl et bs"

  shows "∃phi. wt_method G C pTs rT maxs mxl bs et phi"
proof -
  let ?start = "OK (Some ([],(OK (Class C))#((map OK pTs))@(replicate mxl Err)))
               #(replicate (size bs - 1) (OK None))"

  from wtk obtain maxr r where
    bounded: "check_bounded bs et" and
    result:  "r = kiljvm G maxs maxr rT et bs ?start" and
    success: "∀n < size bs. r!n ≠ Err" and
    instrs:  "0 < size bs" and
    maxr:    "maxr = Suc (length pTs + mxl)"
    by (unfold wt_kil_def) simp

  from bounded have "bounded (exec G maxs rT et bs) (size bs)"
    by (unfold exec_def) (intro bounded_lift check_bounded_is_bounded)
  with wf have bcv:
    "is_bcv (JVMType.le G maxs maxr) Err (exec G maxs rT et bs)
    (size bs) (states G maxs maxr) (kiljvm G maxs maxr rT et bs)"
    by (rule is_bcv_kiljvm)

  from C pTs instrs maxr
  have "?start ∈ list (length bs) (states G maxs maxr)"
    apply (unfold JVM_states_unfold)
    apply (rule listI)
    apply (auto intro: list_appendI dest!: in_set_replicate)
    apply force
    done

  with bcv success result have
    "∃ts∈list (length bs) (states G maxs maxr).
         ?start <=[JVMType.le G maxs maxr] ts ∧
         wt_step (JVMType.le G maxs maxr) Err (exec G maxs rT et bs) ts"
    by (unfold is_bcv_def) auto
  then obtain phi' where
    phi': "phi' ∈ list (length bs) (states G maxs maxr)" and
    s: "?start <=[JVMType.le G maxs maxr] phi'" and
    w: "wt_step (JVMType.le G maxs maxr) Err (exec G maxs rT et bs) phi'"
    by blast
  hence wt_err_step:
    "wt_err_step (sup_state_opt G) (exec G maxs rT et bs) phi'"
    by (simp add: wt_err_step_def exec_def JVM_le_Err_conv)

  from s have le: "JVMType.le G maxs maxr (?start ! 0) (phi'!0)"
    by (drule_tac p=0 in le_listD) (simp add: lesub_def)+

  from phi' have l: "size phi' = size bs" by simp
  with instrs w have "phi' ! 0 ≠ Err" by (unfold wt_step_def) simp
```

```
  with instrs l have phi0: "OK (map ok_val phi' ! 0) = phi' ! 0"
    by (clarsimp simp add: not_Err_eq)

  from phi' have "check_types G maxs maxr phi'" by(simp add: check_types_def)
  also from w have "phi' = map OK (map ok_val phi')"
    by (auto simp add: wt_step_def not_Err_eq intro!: nth_equalityI)
  finally
  have check_types:
    "check_types G maxs maxr (map OK (map ok_val phi'))" .

  from l bounded
  have "bounded (λpc. eff (bs!pc) G pc et) (length phi')"
    by (simp add: exec_def check_bounded_is_bounded)
  hence bounded': "bounded (exec G maxs rT et bs) (length bs)"
    by (auto intro: bounded_lift simp add: exec_def l)
  with wt_err_step
  have "wt_app_eff (sup_state_opt G) (λpc. app (bs!pc) G maxs rT pc et)
                   (λpc. eff (bs!pc) G pc et) (map ok_val phi')"
    by (auto intro: wt_err_imp_wt_app_eff simp add: l exec_def)
  with instrs l le bounded bounded' check_types maxr
  have "wt_method G C pTs rT maxs mxl bs et (map ok_val phi')"
    apply (unfold wt_method_def wt_app_eff_def)
    apply simp
    apply (rule conjI)
     apply (unfold wt_start_def)
     apply (rule JVM_le_convert [THEN iffD1])
     apply (simp (no_asm) add: phi0)
    apply clarify
    apply (erule allE, erule impE, assumption)
    apply (elim conjE)
    apply (clarsimp simp add: lesub_def wt_instr_def)
    apply (simp add: exec_def)
    apply (drule bounded_err_stepD, assumption+)
    apply blast
    done

  thus ?thesis by blast
qed


theorem wt_kil_complete:
  assumes wf:  "wf_prog wf_mb G"
  assumes C:   "is_class G C"
  assumes pTs: "set pTs ⊆ types G"

  assumes wtm: "wt_method G C pTs rT maxs mxl bs et phi"

  shows "wt_kil G C pTs rT maxs mxl et bs"
proof -
  let ?mxr = "1+size pTs+mxl"

  from wtm obtain
    instrs:   "0 < length bs" and
    len:      "length phi = length bs" and
```

```
    bounded:  "check_bounded bs et" and
    ck_types: "check_types G maxs ?mxr (map OK phi)" and
    wt_start: "wt_start G C pTs mxl phi" and
    wt_ins:   "∀pc. pc < length bs ⟶
                   wt_instr (bs ! pc) G rT phi maxs (length bs) et pc"
    by (unfold wt_method_def) simp

  from ck_types len
  have istype_phi:
    "map OK phi ∈ list (length bs) (states G maxs (1+size pTs+mxl))"
    by (auto simp add: check_types_def intro!: listI)

  let ?eff  = "λpc. eff (bs!pc) G pc et"
  let ?app   = "λpc. app (bs!pc) G maxs rT pc et"

  from bounded
  have bounded_exec: "bounded (exec G maxs rT et bs) (size bs)"
    by (unfold exec_def) (intro bounded_lift check_bounded_is_bounded)

  from wt_ins
  have "wt_app_eff (sup_state_opt G) ?app ?eff phi"
    apply (unfold wt_app_eff_def wt_instr_def lesub_def)
    apply (simp (no_asm) only: len)
    apply blast
    done
  with bounded_exec
  have "wt_err_step (sup_state_opt G) (err_step (size phi) ?app ?eff) (map OK phi)"
    by - (erule wt_app_eff_imp_wt_err,simp add: exec_def len)
  hence wt_err:
    "wt_err_step (sup_state_opt G) (exec G maxs rT et bs) (map OK phi)"
    by (unfold exec_def) (simp add: len)

  from wf bounded_exec
  have is_bcv:
    "is_bcv (JVMType.le G maxs ?mxr) Err (exec G maxs rT et bs)
            (size bs) (states G maxs ?mxr) (kiljvm G maxs ?mxr rT et bs)"
    by (rule is_bcv_kiljvm)

  let ?start = "OK (Some ([],(OK (Class C))#((map OK pTs))@(replicate mxl Err)))
                #(replicate (size bs - 1) (OK None))"

  from C pTs instrs
  have start: "?start ∈ list (length bs) (states G maxs ?mxr)"
    apply (unfold JVM_states_unfold)
    apply (rule listI)
    apply (auto intro!: list_appendI dest!: in_set_replicate)
    apply force
    done

  let ?phi = "map OK phi"
  have less_phi: "?start <=[JVMType.le G maxs ?mxr] ?phi"
  proof -
    from len instrs
    have "length ?start = length (map OK phi)" by simp
```

```
    moreover
    { fix n
      from wt_start
      have "G ⊢ ok_val (?start!0) <=' phi!0"
        by (simp add: wt_start_def)
      moreover
      from instrs len
      have "0 < length phi" by simp
      ultimately
      have "JVMType.le G maxs ?mxr (?start!0) (?phi!0)"
        by (simp add: JVM_le_Err_conv Err.le_def lesub_def)
      moreover
      { fix n'
        have "JVMType.le G maxs ?mxr (OK None) (?phi!n)"
          by (auto simp add: JVM_le_Err_conv Err.le_def lesub_def
            split: err.splits)
        hence "⟦ n = Suc n'; n < length ?start ⟧
          ⟹ JVMType.le G maxs ?mxr (?start!n) (?phi!n)"
          by simp
      }
      ultimately
      have "n < length ?start ⟹ (?start!n) <=_(JVMType.le G maxs ?mxr) (?phi!n)"
        by (unfold lesub_def) (cases n, blast+)
    }
    ultimately show ?thesis by (rule le_listI)
  qed

  from wt_err
  have "wt_step (JVMType.le G maxs ?mxr) Err (exec G maxs rT et bs) ?phi"
    by (simp add: wt_err_step_def JVM_le_Err_conv)
  with start istype_phi less_phi is_bcv
  have "∀p. p < length bs ⟶ kiljvm G maxs ?mxr rT et bs ?start ! p ≠ Err"
    by (unfold is_bcv_def) auto
  with bounded instrs
  show "wt_kil G C pTs rT maxs mxl et bs" by (unfold wt_kil_def) simp
qed


theorem jvm_kildall_sound_complete:
  "wt_jvm_prog_kildall G = (∃Phi. wt_jvm_prog G Phi)"
proof
  let ?Phi = "λC sig. let (C,rT,(maxs,maxl,ins,et)) = the (method (G,C) sig) in
              SOME phi. wt_method G C (snd sig) rT maxs maxl ins et phi"

  assume "wt_jvm_prog_kildall G"
  hence "wt_jvm_prog G ?Phi"
    apply (unfold wt_jvm_prog_def wt_jvm_prog_kildall_def)
    apply (erule jvm_prog_lift)
    apply (auto dest!: wt_kil_correct intro: someI)
    done
  thus "∃Phi. wt_jvm_prog G Phi" by fast
next
  assume "∃Phi. wt_jvm_prog G Phi"
  thus "wt_jvm_prog_kildall G"
```

```
  apply (clarify)
  apply (unfold wt_jvm_prog_def wt_jvm_prog_kildall_def)
  apply (erule jvm_prog_lift)
  apply (auto intro: wt_kil_complete)
  done
qed

end
```

Operations on lists beyond the standard List theory

```
theory More_List
imports Main
begin

hide_const (open) Finite_Set.fold
```

Repairing code generator setup

```
declare (in lattice) Inf_fin_set_fold [code_unfold del]
declare (in lattice) Sup_fin_set_fold [code_unfold del]
declare (in linorder) Min_fin_set_fold [code_unfold del]
declare (in linorder) Max_fin_set_fold [code_unfold del]
declare (in complete_lattice) Inf_set_fold [code_unfold del]
declare (in complete_lattice) Sup_set_fold [code_unfold del]
declare rev_foldl_cons [code del]
```

Fold combinator with canonical argument order

```
primrec fold :: "('a ⇒ 'b ⇒ 'b) ⇒ 'a list ⇒ 'b ⇒ 'b" where
    "fold f [] = id"
  | "fold f (x # xs) = fold f xs ∘ f x"

lemma foldl_fold:
  "foldl f s xs = fold (λx s. f s x) xs s"
  by (induct xs arbitrary: s) simp_all

lemma foldr_fold_rev:
  "foldr f xs = fold f (rev xs)"
  by (simp add: foldr_foldl foldl_fold expand_fun_eq)

lemma fold_rev_conv [code_unfold]:
  "fold f (rev xs) = foldr f xs"
  by (simp add: foldr_fold_rev)

lemma fold_cong [fundef_cong, recdef_cong]:
  "a = b ⟹ xs = ys ⟹ (⋀x. x ∈ set xs ⟹ f x = g x)
    ⟹ fold f xs a = fold g ys b"
  by (induct ys arbitrary: a b xs) simp_all

lemma fold_id:
  assumes "⋀x. x ∈ set xs ⟹ f x = id"
  shows "fold f xs = id"
  using assms by (induct xs) simp_all

lemma fold_apply:
  assumes "⋀x. x ∈ set xs ⟹ h ∘ g x = f x ∘ h"
  shows "h ∘ fold g xs = fold f xs ∘ h"
```

```
  using assms by (induct xs) (simp_all add: expand_fun_eq)

lemma fold_invariant:
  assumes "⋀x. x ∈ set xs ⟹ Q x" and "P s"
    and "⋀x s. Q x ⟹ P s ⟹ P (f x s)"
  shows "P (fold f xs s)"
  using assms by (induct xs arbitrary: s) simp_all

lemma fold_weak_invariant:
  assumes "P s"
    and "⋀s x. x ∈ set xs ⟹ P s ⟹ P (f x s)"
  shows "P (fold f xs s)"
  using assms by (induct xs arbitrary: s) simp_all

lemma fold_append [simp]:
  "fold f (xs @ ys) = fold f ys ∘ fold f xs"
  by (induct xs) simp_all

lemma fold_map [code_unfold]:
  "fold g (map f xs) = fold (g o f) xs"
  by (induct xs) simp_all

lemma fold_rev:
  assumes "⋀x y. x ∈ set xs ⟹ y ∈ set xs ⟹ f y ∘ f x = f x ∘ f y"
  shows "fold f (rev xs) = fold f xs"
  using assms by (induct xs) (simp_all del: o_apply add: fold_apply)

lemma foldr_fold:
  assumes "⋀x y. x ∈ set xs ⟹ y ∈ set xs ⟹ f y ∘ f x = f x ∘ f y"
  shows "foldr f xs = fold f xs"
  using assms unfolding foldr_fold_rev by (rule fold_rev)

lemma fold_Cons_rev:
  "fold Cons xs = append (rev xs)"
  by (induct xs) simp_all

lemma rev_conv_fold [code]:
  "rev xs = fold Cons xs []"
  by (simp add: fold_Cons_rev)

lemma fold_append_concat_rev:
  "fold append xss = append (concat (rev xss))"
  by (induct xss) simp_all

lemma concat_conv_foldr [code]:
  "concat xss = foldr append xss []"
  by (simp add: fold_append_concat_rev foldr_fold_rev)

lemma fold_plus_listsum_rev:
  "fold plus xs = plus (listsum (rev xs))"
  by (induct xs) (simp_all add: add.assoc)

lemma listsum_conv_foldr [code]:
  "listsum xs = foldr plus xs 0"
```

```
  by (fact listsum_foldr)

lemma sort_key_conv_fold:
  assumes "inj_on f (set xs)"
  shows "sort_key f xs = fold (insort_key f) xs []"
proof -
  have "fold (insort_key f) (rev xs) = fold (insort_key f) xs"
  proof (rule fold_rev, rule ext)
    fix zs
    fix x y
    assume "x ∈ set xs" "y ∈ set xs"
    with assms have *: "f y = f x ⟹ y = x" by (auto dest: inj_onD)
    show "(insort_key f y ∘ insort_key f x) zs = (insort_key f x ∘ insort_key f y) zs"
      by (induct zs) (auto dest: *)
  qed
  then show ?thesis by (simp add: sort_key_def foldr_fold_rev)
qed

lemma sort_conv_fold:
  "sort xs = fold insort xs []"
  by (rule sort_key_conv_fold) simp
```

Finite_Set.fold and *fold*

```
lemma (in fun_left_comm) fold_set_remdups:
  "Finite_Set.fold f y (set xs) = fold f (remdups xs) y"
  by (rule sym, induct xs arbitrary: y) (simp_all add: fold_fun_comm insert_absorb)

lemma (in fun_left_comm_idem) fold_set:
  "Finite_Set.fold f y (set xs) = fold f xs y"
  by (rule sym, induct xs arbitrary: y) (simp_all add: fold_fun_comm)

lemma (in ab_semigroup_idem_mult) fold1_set:
  assumes "xs ≠ []"
  shows "Finite_Set.fold1 times (set xs) = fold times (tl xs) (hd xs)"
proof -
  interpret fun_left_comm_idem times by (fact fun_left_comm_idem)
  from assms obtain y ys where xs: "xs = y # ys"
    by (cases xs) auto
  show ?thesis
  proof (cases "set ys = {}")
    case True with xs show ?thesis by simp
  next
    case False
    then have "fold1 times (insert y (set ys)) = Finite_Set.fold times y (set ys)"
      by (simp only: finite_set fold1_eq_fold_idem)
    with xs show ?thesis by (simp add: fold_set mult_commute)
  qed
qed

lemma (in lattice) Inf_fin_set_fold:
  "Inf_fin (set (x # xs)) = fold inf xs x"
proof -
  interpret ab_semigroup_idem_mult "inf :: 'a ⇒ 'a ⇒ 'a"
    by (fact ab_semigroup_idem_mult_inf)
```

```
  show ?thesis
    by (simp add: Inf_fin_def fold1_set del: set.simps)
qed

lemma (in lattice) Inf_fin_set_foldr [code_unfold]:
  "Inf_fin (set (x # xs)) = foldr inf xs x"
  by (simp add: Inf_fin_set_fold ac_simps foldr_fold expand_fun_eq del: set.simps)

lemma (in lattice) Sup_fin_set_fold:
  "Sup_fin (set (x # xs)) = fold sup xs x"
proof -
  interpret ab_semigroup_idem_mult "sup :: 'a ⇒ 'a ⇒ 'a"
    by (fact ab_semigroup_idem_mult_sup)
  show ?thesis
    by (simp add: Sup_fin_def fold1_set del: set.simps)
qed

lemma (in lattice) Sup_fin_set_foldr [code_unfold]:
  "Sup_fin (set (x # xs)) = foldr sup xs x"
  by (simp add: Sup_fin_set_fold ac_simps foldr_fold expand_fun_eq del: set.simps)

lemma (in linorder) Min_fin_set_fold:
  "Min (set (x # xs)) = fold min xs x"
proof -
  interpret ab_semigroup_idem_mult "min :: 'a ⇒ 'a ⇒ 'a"
    by (fact ab_semigroup_idem_mult_min)
  show ?thesis
    by (simp add: Min_def fold1_set del: set.simps)
qed

lemma (in linorder) Min_fin_set_foldr [code_unfold]:
  "Min (set (x # xs)) = foldr min xs x"
  by (simp add: Min_fin_set_fold ac_simps foldr_fold expand_fun_eq del: set.simps)

lemma (in linorder) Max_fin_set_fold:
  "Max (set (x # xs)) = fold max xs x"
proof -
  interpret ab_semigroup_idem_mult "max :: 'a ⇒ 'a ⇒ 'a"
    by (fact ab_semigroup_idem_mult_max)
  show ?thesis
    by (simp add: Max_def fold1_set del: set.simps)
qed

lemma (in linorder) Max_fin_set_foldr [code_unfold]:
  "Max (set (x # xs)) = foldr max xs x"
  by (simp add: Max_fin_set_fold ac_simps foldr_fold expand_fun_eq del: set.simps)

lemma (in complete_lattice) Inf_set_fold:
  "Inf (set xs) = fold inf xs top"
proof -
  interpret fun_left_comm_idem "inf :: 'a ⇒ 'a ⇒ 'a"
    by (fact fun_left_comm_idem_inf)
  show ?thesis by (simp add: Inf_fold_inf fold_set inf_commute)
qed
```

lemma *(***in** *complete_lattice) Inf_set_foldr [code_unfold]:*
  "Inf (set xs) = foldr inf xs top"
  by *(simp add: Inf_set_fold ac_simps foldr_fold expand_fun_eq)*

lemma *(***in** *complete_lattice) Sup_set_fold:*
  "Sup (set xs) = fold sup xs bot"
proof -
  interpret *fun_left_comm_idem "sup :: 'a ⇒ 'a ⇒ 'a"*
    by *(fact fun_left_comm_idem_sup)*
  show *?thesis* **by** *(simp add: Sup_fold_sup fold_set sup_commute)*
qed

lemma *(***in** *complete_lattice) Sup_set_foldr [code_unfold]:*
  "Sup (set xs) = foldr sup xs bot"
  by *(simp add: Sup_set_fold ac_simps foldr_fold expand_fun_eq)*

lemma *(***in** *complete_lattice) INFI_set_fold:*
  "INFI (set xs) f = fold (inf ∘ f) xs top"
  unfolding *INFI_def set_map [symmetric] Inf_set_fold fold_map* ..

lemma *(***in** *complete_lattice) SUPR_set_fold:*
  "SUPR (set xs) f = fold (sup ∘ f) xs bot"
  unfolding *SUPR_def set_map [symmetric] Sup_set_fold fold_map* ..

nth_map

definition *nth_map :: "nat ⇒ ('a ⇒ 'a) ⇒ 'a list ⇒ 'a list"* **where**
  "nth_map n f xs = (if n < length xs then
      take n xs @ [f (xs ! n)] @ drop (Suc n) xs
    else xs)"

lemma *nth_map_id:*
  "n ≥ length xs ⟹ nth_map n f xs = xs"
  by *(simp add: nth_map_def)*

lemma *nth_map_unfold:*
  "n < length xs ⟹ nth_map n f xs = take n xs @ [f (xs ! n)] @ drop (Suc n) xs"
  by *(simp add: nth_map_def)*

lemma *nth_map_Nil [simp]:*
  "nth_map n f [] = []"
  by *(simp add: nth_map_def)*

lemma *nth_map_zero [simp]:*
  "nth_map 0 f (x # xs) = f x # xs"
  by *(simp add: nth_map_def)*

lemma *nth_map_Suc [simp]:*
  "nth_map (Suc n) f (x # xs) = x # nth_map n f xs"
  by *(simp add: nth_map_def)*

end

Relating (finite) sets and lists **theory** *More_Set*

```
imports Main More_List
begin
```

4.23.1 Various additional set functions

```
definition is_empty :: "'a set ⇒ bool" where
  "is_empty A ⟷ A = {}"

definition remove :: "'a ⇒ 'a set ⇒ 'a set" where
  "remove x A = A - {x}"

lemma fun_left_comm_idem_remove:
  "fun_left_comm_idem remove"
proof -
  have rem: "remove = (λx A. A - {x})" by (simp add: expand_fun_eq remove_def)
  show ?thesis by (simp only: fun_left_comm_idem_remove rem)
qed

lemma minus_fold_remove:
  assumes "finite A"
  shows "B - A = Finite_Set.fold remove B A"
proof -
  have rem: "remove = (λx A. A - {x})" by (simp add: expand_fun_eq remove_def)
  show ?thesis by (simp only: rem assms minus_fold_remove)
qed

definition project :: "('a ⇒ bool) ⇒ 'a set ⇒ 'a set" where
  "project P A = {a∈A. P a}"
```

4.23.2 Basic set operations

```
lemma is_empty_set:
  "is_empty (set xs) ⟷ null xs"
  by (simp add: is_empty_def null_empty)

lemma ball_set:
  "(∀x∈set xs. P x) ⟷ list_all P xs"
  by (rule list_ball_code)

lemma bex_set:
  "(∃x∈set xs. P x) ⟷ list_ex P xs"
  by (rule list_bex_code)

lemma empty_set:
  "{} = set []"
  by simp

lemma insert_set_compl:
  "insert x (- set xs) = - set (removeAll x xs)"
  by auto

lemma remove_set_compl:
  "remove x (- set xs) = - set (List.insert x xs)"
  by (auto simp del: mem_def simp add: remove_def List.insert_def)
```

```
lemma image_set:
  "image f (set xs) = set (map f xs)"
  by simp

lemma project_set:
  "project P (set xs) = set (filter P xs)"
  by (auto simp add: project_def)
```

4.23.3 Functorial set operations

```
lemma union_set:
  "set xs ∪ A = fold Set.insert xs A"
proof -
  interpret fun_left_comm_idem Set.insert
    by (fact fun_left_comm_idem_insert)
  show ?thesis by (simp add: union_fold_insert fold_set)
qed

lemma union_set_foldr:
  "set xs ∪ A = foldr Set.insert xs A"
proof -
  have "⋀x y :: 'a. insert y ∘ insert x = insert x ∘ insert y"
    by (auto intro: ext)
  then show ?thesis by (simp add: union_set foldr_fold)
qed

lemma minus_set:
  "A - set xs = fold remove xs A"
proof -
  interpret fun_left_comm_idem remove
    by (fact fun_left_comm_idem_remove)
  show ?thesis
    by (simp add: minus_fold_remove [of _ A] fold_set)
qed

lemma minus_set_foldr:
  "A - set xs = foldr remove xs A"
proof -
  have "⋀x y :: 'a. remove y ∘ remove x = remove x ∘ remove y"
    by (auto simp add: remove_def intro: ext)
  then show ?thesis by (simp add: minus_set foldr_fold)
qed
```

4.23.4 Derived set operations

```
lemma member:
  "a ∈ A ⟷ (∃x∈A. a = x)"
  by simp

lemma subset_eq:
  "A ⊆ B ⟷ (∀x∈A. x ∈ B)"
  by (fact subset_eq)
```

lemma *subset:*
  "A ⊂ B ⟷ A ⊆ B ∧ ¬ B ⊆ A"
  by *(fact less_le_not_le)*

lemma *set_eq:*
  "A = B ⟷ A ⊆ B ∧ B ⊆ A"
  by *(fact eq_iff)*

lemma *inter:*
  "A ∩ B = project (λx. x ∈ A) B"
  by *(auto simp add: project_def)*

4.23.5 Various lemmas

lemma *not_set_compl:*
  "Not ∘ set xs = - set xs"
  by *(simp add: fun_Compl_def bool_Compl_def comp_def expand_fun_eq)*

end

A crude implementation of finite sets by lists -- avoid using this at any cost! **theory** *Executable_Set*
imports *More_Set*
begin

declare *mem_def [code del]*
declare *Collect_def [code del]*
declare *insert_code [code del]*
declare *vimage_code [code del]*

4.23.6 Set representation

setup *{**
  Code.add_type_cmd "set"
**}*

definition *Set :: "'a list ⇒ 'a set"* **where**
  [simp]: "Set = set"

definition *Coset :: "'a list ⇒ 'a set"* **where**
  [simp]: "Coset xs = - set xs"

setup *{**
  Code.add_signature_cmd ("Set", "'a list ⇒ 'a set")
  #> Code.add_signature_cmd ("Coset", "'a list ⇒ 'a set")
  #> Code.add_signature_cmd ("set", "'a list ⇒ 'a set")
  #> Code.add_signature_cmd ("op ∈", "'a ⇒ 'a set ⇒ bool")
**}*

code_datatype *Set Coset*

consts_code
  *Coset ("⟨**module**⟩Coset")*
  *Set ("⟨**module**⟩Set")*

```
attach {*
  datatype 'a set = Set of 'a list | Coset of 'a list;
*} — This assumes that there won't be a Coset without a Set
```

4.23.7 Basic operations

```
lemma [code]:
  "set xs = Set (remdups xs)"
  by simp

lemma [code]:
  "x ∈ Set xs ⟷ member xs x"
  "x ∈ Coset xs ⟷ ¬ member xs x"
  by (simp_all add: mem_iff)

definition is_empty :: "'a set ⇒ bool" where
  [simp]: "is_empty A ⟷ A = {}"

lemma [code_unfold]:
  "A = {} ⟷ is_empty A"
  by simp

definition empty :: "'a set" where
  [simp]: "empty = {}"

lemma [code_unfold]:
  "{} = empty"
  by simp

lemma [code_unfold, code_inline del]:
  "empty = Set []"
  by simp — Otherwise η-expansion produces funny things.

setup {*
  Code.add_signature_cmd ("is_empty", "'a set ⇒ bool")
  #> Code.add_signature_cmd ("empty", "'a set")
  #> Code.add_signature_cmd ("insert", "'a ⇒ 'a set ⇒ 'a set")
  #> Code.add_signature_cmd ("More_Set.remove", "'a ⇒ 'a set ⇒ 'a set")
  #> Code.add_signature_cmd ("image", "('a ⇒ 'b) ⇒ 'a set ⇒ 'b set")
  #> Code.add_signature_cmd ("More_Set.project", "('a ⇒ bool) ⇒ 'a set ⇒ 'a set")
  #> Code.add_signature_cmd ("Ball", "'a set ⇒ ('a ⇒ bool) ⇒ bool")
  #> Code.add_signature_cmd ("Bex", "'a set ⇒ ('a ⇒ bool) ⇒ bool")
  #> Code.add_signature_cmd ("card", "'a set ⇒ nat")
*}

lemma is_empty_Set [code]:
  "is_empty (Set xs) ⟷ null xs"
  by (simp add: empty_null)

lemma empty_Set [code]:
  "empty = Set []"
  by simp

lemma insert_Set [code]:
```

```
  "insert x (Set xs) = Set (List.insert x xs)"
  "insert x (Coset xs) = Coset (removeAll x xs)"
  by (simp_all add: set_insert)

lemma remove_Set [code]:
  "remove x (Set xs) = Set (removeAll x xs)"
  "remove x (Coset xs) = Coset (List.insert x xs)"
  by (auto simp add: set_insert remove_def)

lemma image_Set [code]:
  "image f (Set xs) = Set (remdups (map f xs))"
  by simp

lemma project_Set [code]:
  "project P (Set xs) = Set (filter P xs)"
  by (simp add: project_set)

lemma Ball_Set [code]:
  "Ball (Set xs) P ⟷ list_all P xs"
  by (simp add: ball_set)

lemma Bex_Set [code]:
  "Bex (Set xs) P ⟷ list_ex P xs"
  by (simp add: bex_set)

lemma card_Set [code]:
  "card (Set xs) = length (remdups xs)"
proof -
  have "card (set (remdups xs)) = length (remdups xs)"
    by (rule distinct_card) simp
  then show ?thesis by simp
qed
```

4.23.8 Derived operations

```
definition set_eq :: "'a set ⇒ 'a set ⇒ bool" where
  [simp]: "set_eq = op ="

lemma [code_unfold]:
  "op = = set_eq"
  by simp

definition subset_eq :: "'a set ⇒ 'a set ⇒ bool" where
  [simp]: "subset_eq = op ⊆"

lemma [code_unfold]:
  "op ⊆ = subset_eq"
  by simp

definition subset :: "'a set ⇒ 'a set ⇒ bool" where
  [simp]: "subset = op ⊂"

lemma [code_unfold]:
  "op ⊂ = subset"
```

```
  by simp

setup {*
  Code.add_signature_cmd ("set_eq", "'a set ⇒ 'a set ⇒ bool")
  #> Code.add_signature_cmd ("subset_eq", "'a set ⇒ 'a set ⇒ bool")
  #> Code.add_signature_cmd ("subset", "'a set ⇒ 'a set ⇒ bool")
*}

lemma set_eq_subset_eq [code]:
  "set_eq A B ⟷ subset_eq A B ∧ subset_eq B A"
  by auto

lemma subset_eq_forall [code]:
  "subset_eq A B ⟷ (∀x∈A. x ∈ B)"
  by (simp add: subset_eq)

lemma subset_subset_eq [code]:
  "subset A B ⟷ subset_eq A B ∧ ¬ subset_eq B A"
  by (simp add: subset)
```

4.23.9 Functorial operations

```
definition inter :: "'a set ⇒ 'a set ⇒ 'a set" where
  [simp]: "inter = op ∩"

lemma [code_unfold]:
  "op ∩ = inter"
  by simp

definition subtract :: "'a set ⇒ 'a set ⇒ 'a set" where
  [simp]: "subtract A B = B - A"

lemma [code_unfold]:
  "B - A = subtract A B"
  by simp

definition union :: "'a set ⇒ 'a set ⇒ 'a set" where
  [simp]: "union = op ∪"

lemma [code_unfold]:
  "op ∪ = union"
  by simp

definition Inf :: "'a::complete_lattice set ⇒ 'a" where
  [simp]: "Inf = Complete_Lattice.Inf"

lemma [code_unfold]:
  "Complete_Lattice.Inf = Inf"
  by simp

definition Sup :: "'a::complete_lattice set ⇒ 'a" where
  [simp]: "Sup = Complete_Lattice.Sup"

lemma [code_unfold]:
```

```
  "Complete_Lattice.Sup = Sup"
  by simp

definition Inter :: "'a set set ⇒ 'a set" where
  [simp]: "Inter = Inf"

lemma [code_unfold]:
  "Inf = Inter"
  by simp

definition Union :: "'a set set ⇒ 'a set" where
  [simp]: "Union = Sup"

lemma [code_unfold]:
  "Sup = Union"
  by simp

setup {*
  Code.add_signature_cmd ("inter", "'a set ⇒ 'a set ⇒ 'a set")
  #> Code.add_signature_cmd ("subtract", "'a set ⇒ 'a set ⇒ 'a set")
  #> Code.add_signature_cmd ("union", "'a set ⇒ 'a set ⇒ 'a set")
  #> Code.add_signature_cmd ("Inf", "'a set ⇒ 'a")
  #> Code.add_signature_cmd ("Sup", "'a set ⇒ 'a")
  #> Code.add_signature_cmd ("Inter", "'a set set ⇒ 'a set")
  #> Code.add_signature_cmd ("Union", "'a set set ⇒ 'a set")
*}

lemma inter_project [code]:
  "inter A (Set xs) = Set (List.filter (λx. x ∈ A) xs)"
  "inter A (Coset xs) = foldr remove xs A"
  by (simp add: inter project_def) (simp add: Diff_eq [symmetric] minus_set_foldr)

lemma subtract_remove [code]:
  "subtract (Set xs) A = foldr remove xs A"
  "subtract (Coset xs) A = Set (List.filter (λx. x ∈ A) xs)"
  by (auto simp add: minus_set_foldr)

lemma union_insert [code]:
  "union (Set xs) A = foldr insert xs A"
  "union (Coset xs) A = Coset (List.filter (λx. x ∉ A) xs)"
  by (auto simp add: union_set_foldr)

lemma Inf_inf [code]:
  "Inf (Set xs) = foldr inf xs (top :: 'a::complete_lattice)"
  "Inf (Coset []) = (bot :: 'a::complete_lattice)"
  by (simp_all add: Inf_UNIV Inf_set_foldr)

lemma Sup_sup [code]:
  "Sup (Set xs) = foldr sup xs (bot :: 'a::complete_lattice)"
  "Sup (Coset []) = (top :: 'a::complete_lattice)"
  by (simp_all add: Sup_UNIV Sup_set_foldr)

lemma Inter_inter [code]:
  "Inter (Set xs) = foldr inter xs (Coset [])"
```

```
  "Inter (Coset []) = empty"
  unfolding Inter_def Inf_inf by simp_all

lemma Union_union [code]:
  "Union (Set xs) = foldr union xs empty"
  "Union (Coset []) = Coset []"
  unfolding Union_def Sup_sup by simp_all

hide_const (open) is_empty empty remove
  set_eq subset_eq subset inter union subtract Inf Sup Inter Union

end
```

4.24 Example Welltypings

```
theory BVExample
imports "../JVM/JVMListExample" BVSpecTypeSafe JVM Executable_Set
begin
```

This theory shows type correctness of the example program in section 3.6 (p. 83) by explicitly providing a welltyping. It also shows that the start state of the program conforms to the welltyping; hence type safe execution is guaranteed.

4.24.1 Setup

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

```
axioms
  distinct_classes: "list_nam ≠ test_nam"
  distinct_fields:  "val_nam ≠ next_nam"
```

Abbreviations for definitions we will have to use often in the proofs below:

```
lemmas name_defs   = list_name_def test_name_def val_name_def next_name_def
lemmas system_defs = SystemClasses_def ObjectC_def NullPointerC_def
                     OutOfMemoryC_def ClassCastC_def
lemmas class_defs  = list_class_def test_class_def
```

These auxiliary proofs are for efficiency: class lookup, subclass relation, method and field lookup are computed only once:

```
lemma class_Object [simp]:
  "class E Object = Some (undefined, [],[])"
  by (simp add: class_def system_defs E_def)

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

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

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

lemma class_list [simp]:
  "class E list_name = Some list_class"
  by (simp add: class_def system_defs E_def name_defs distinct_classes [symmetric])

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

lemma E_classes [simp]:
  "{C. is_class E C} = {list_name, test_name, Xcpt NullPointer,
```

```
                           Xcpt ClassCast, Xcpt OutOfMemory, Object}"
  by (auto simp add: is_class_def class_def system_defs E_def name_defs class_defs)
```

The subclass releation spelled out:

```
lemma subcls1:
  "subcls1 E = {(list_name,Object), (test_name,Object), (Xcpt NullPointer, Object),
               (Xcpt ClassCast, Object), (Xcpt OutOfMemory, Object)}"
apply (simp add: subcls1_def2)
apply (simp add: name_defs class_defs system_defs E_def class_def)
apply (simp add: Sigma_def)
apply auto
done
```

The subclass relation is acyclic; hence its converse is well founded:

```
lemma notin_rtrancl:
  "(a, b) ∈ r* ⟹ a ≠ b ⟹ (⋀y. (a, y) ∉ r) ⟹ False"
  by (auto elim: converse_rtranclE)

lemma acyclic_subcls1_E: "acyclic (subcls1 E)"
  apply (rule acyclicI)
  apply (simp add: subcls1)
  apply (auto dest!: tranclD)
  apply (auto elim!: notin_rtrancl simp add: name_defs distinct_classes)
  done

lemma wf_subcls1_E: "wf ((subcls1 E)^-1)"
  apply (rule finite_acyclic_wf_converse)
  apply (simp add: subcls1 del: insert_iff)
  apply (rule acyclic_subcls1_E)
  done
```

Method and field lookup:

```
lemma method_Object [simp]:
  "method (E, Object) = Map.empty"
  by (simp add: method_rec_lemma [OF class_Object wf_subcls1_E])

lemma method_append [simp]:
  "method (E, list_name) (append_name, [Class list_name]) =
  Some (list_name, PrimT Void, 3, 0, append_ins, [(1, 2, 8, Xcpt NullPointer)])"
  apply (insert class_list)
  apply (unfold list_class_def)
  apply (drule method_rec_lemma [OF _ wf_subcls1_E])
  apply simp
  done

lemma method_makelist [simp]:
  "method (E, test_name) (makelist_name, []) =
  Some (test_name, PrimT Void, 3, 2, make_list_ins, [])"
  apply (insert class_test)
  apply (unfold test_class_def)
  apply (drule method_rec_lemma [OF _ wf_subcls1_E])
  apply simp
  done
```

```
lemma field_val [simp]:
  "field (E, list_name) val_name = Some (list_name, PrimT Integer)"
  apply (unfold TypeRel.field_def)
  apply (insert class_list)
  apply (unfold list_class_def)
  apply (drule fields_rec_lemma [OF _ wf_subcls1_E])
  apply simp
  done

lemma field_next [simp]:
  "field (E, list_name) next_name = Some (list_name, Class list_name)"
  apply (unfold TypeRel.field_def)
  apply (insert class_list)
  apply (unfold list_class_def)
  apply (drule fields_rec_lemma [OF _ wf_subcls1_E])
  apply (simp add: name_defs distinct_fields [symmetric])
  done

lemma [simp]: "fields (E, Object) = []"
   by (simp add: fields_rec_lemma [OF class_Object wf_subcls1_E])

lemma [simp]: "fields (E, Xcpt NullPointer) = []"
  by (simp add: fields_rec_lemma [OF class_NullPointer wf_subcls1_E])

lemma [simp]: "fields (E, Xcpt ClassCast) = []"
  by (simp add: fields_rec_lemma [OF class_ClassCast wf_subcls1_E])

lemma [simp]: "fields (E, Xcpt OutOfMemory) = []"
  by (simp add: fields_rec_lemma [OF class_OutOfMemory wf_subcls1_E])

lemma [simp]: "fields (E, test_name) = []"
  apply (insert class_test)
  apply (unfold test_class_def)
  apply (drule fields_rec_lemma [OF _ wf_subcls1_E])
  apply simp
  done

lemmas [simp] = is_class_def
```

The next definition and three proof rules implement an algorithm to enumarate natural numbers. The command `apply (elim pc_end pc_next pc_0` transforms a goal of the form

pc < n $\Longrightarrow$ *P pc*

into a series of goals

P (0::'a)

P (Suc 0)

...

P n

```
definition intervall :: "nat ⇒ nat ⇒ nat ⇒ bool" ("_ ∈ [_, _')") where
  "x ∈ [a, b) ≡ a ≤ x ∧ x < b"

lemma pc_0: "x < n ⟹ (x ∈ [0, n) ⟹ P x) ⟹ P x"
  by (simp add: intervall_def)

lemma pc_next: "x ∈ [n0, n) ⟹ P n0 ⟹ (x ∈ [Suc n0, n) ⟹ P x) ⟹ P x"
  apply (cases "x=n0")
  apply (auto simp add: intervall_def)
  done

lemma pc_end: "x ∈ [n,n) ⟹ P x"
  by (unfold intervall_def) arith
```

4.24.2 Program structure

The program is structurally wellformed:

```
lemma wf_struct:
  "wf_prog (λG C mb. True) E" (is "wf_prog ?mb E")
proof -
  have "unique E"
    by (simp add: system_defs E_def class_defs name_defs distinct_classes)
  moreover
  have "set SystemClasses ⊆ set E" by (simp add: system_defs E_def)
  hence "wf_syscls E" by (rule wf_syscls)
  moreover
  have "wf_cdecl ?mb E ObjectC" by (simp add: wf_cdecl_def ObjectC_def)
  moreover
  have "wf_cdecl ?mb E NullPointerC"
    by (auto elim: notin_rtrancl
            simp add: wf_cdecl_def name_defs NullPointerC_def subcls1)
  moreover
  have "wf_cdecl ?mb E ClassCastC"
    by (auto elim: notin_rtrancl
            simp add: wf_cdecl_def name_defs ClassCastC_def subcls1)
  moreover
  have "wf_cdecl ?mb E OutOfMemoryC"
    by (auto elim: notin_rtrancl
            simp add: wf_cdecl_def name_defs OutOfMemoryC_def subcls1)
  moreover
  have "wf_cdecl ?mb E (list_name, list_class)"
    apply (auto elim!: notin_rtrancl
            simp add: wf_cdecl_def wf_fdecl_def list_class_def
                      wf_mdecl_def wf_mhead_def subcls1)
    apply (auto simp add: name_defs distinct_classes distinct_fields)
    done
  moreover
  have "wf_cdecl ?mb E (test_name, test_class)"
    apply (auto elim!: notin_rtrancl
            simp add: wf_cdecl_def wf_fdecl_def test_class_def
                      wf_mdecl_def wf_mhead_def subcls1)
    apply (auto simp add: name_defs distinct_classes distinct_fields)
    done
  ultimately
```

```
  show ?thesis
    by (simp add: wf_prog_def ws_prog_def wf_cdecl_mrT_cdecl_mdecl E_def SystemClasses_def)
qed
```

4.24.3 Welltypings

We show welltypings of the methods *append_name* in class *list_name*, and *makelist_name* in class *test_name*:

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

definition phi_append :: method_type ("φa") where
  "φa ≡ map (λ(x,y). Some (x, map OK y)) [
   (                                   [], [Class list_name, Class list_name]),
   (                     [Class list_name], [Class list_name, Class list_name]),
   (                     [Class list_name], [Class list_name, Class list_name]),
   (    [Class list_name, Class list_name], [Class list_name, Class list_name]),
   ([NT, Class list_name, Class list_name], [Class list_name, Class list_name]),
   (                     [Class list_name], [Class list_name, Class list_name]),
   (    [Class list_name, Class list_name], [Class list_name, Class list_name]),
   (                          [PrimT Void], [Class list_name, Class list_name]),
   (                        [Class Object], [Class list_name, Class list_name]),
   (                                   [], [Class list_name, Class list_name]),
   (                     [Class list_name], [Class list_name, Class list_name]),
   (    [Class list_name, Class list_name], [Class list_name, Class list_name]),
   (                                   [], [Class list_name, Class list_name]),
   (                          [PrimT Void], [Class list_name, Class list_name])]"


lemma bounded_append [simp]:
  "check_bounded append_ins [(Suc 0, 2, 8, Xcpt NullPointer)]"
  apply (simp add: check_bounded_def)
  apply (simp add: nat_number append_ins_def)
  apply (rule allI, rule impI)
  apply (elim pc_end pc_next pc_0)
  apply auto
  done

lemma types_append [simp]: "check_types E 3 (Suc (Suc 0)) (map OK φa)"
  apply (auto simp add: check_types_def phi_append_def JVM_states_unfold)
  apply (unfold list_def)
  apply auto
  done

lemma wt_append [simp]:
  "wt_method E list_name [Class list_name] (PrimT Void) 3 0 append_ins
            [(Suc 0, 2, 8, Xcpt NullPointer)] φa"
  apply (simp add: wt_method_def wt_start_def wt_instr_def)
  apply (simp add: phi_append_def append_ins_def)
  apply clarify
  apply (elim pc_end pc_next pc_0)
  apply simp
  apply (fastsimp simp add: match_exception_entry_def sup_state_conv subcls1)
```

```
  apply simp
  apply simp
  apply (fastsimp simp add: sup_state_conv subcls1)
  apply simp
  apply (simp add: app_def xcpt_app_def)
  apply simp
  apply simp
  apply simp
  apply (simp add: match_exception_entry_def)
  apply (simp add: match_exception_entry_def)
  apply simp
  apply simp
  done
```

Some abbreviations for readability

```
abbreviation Clist :: ty
  where "Clist == Class list_name"
abbreviation Ctest :: ty
  where "Ctest == Class test_name"

definition phi_makelist :: method_type ("φm") where
  "φm ≡ map (λ(x,y). Some (x, y)) [
    (                                  [], [OK Ctest, Err     , Err     ]),
    (                             [Clist], [OK Ctest, Err     , Err     ]),
    (                      [Clist, Clist], [OK Ctest, Err     , Err     ]),
    (                             [Clist], [OK Clist, Err     , Err     ]),
    (              [PrimT Integer, Clist], [OK Clist, Err     , Err     ]),
    (                                  [], [OK Clist, Err     , Err     ]),
    (                             [Clist], [OK Clist, Err     , Err     ]),
    (                      [Clist, Clist], [OK Clist, Err     , Err     ]),
    (                             [Clist], [OK Clist, OK Clist, Err     ]),
    (              [PrimT Integer, Clist], [OK Clist, OK Clist, Err     ]),
    (                                  [], [OK Clist, OK Clist, Err     ]),
    (                             [Clist], [OK Clist, OK Clist, Err     ]),
    (                      [Clist, Clist], [OK Clist, OK Clist, Err     ]),
    (                             [Clist], [OK Clist, OK Clist, OK Clist]),
    (              [PrimT Integer, Clist], [OK Clist, OK Clist, OK Clist]),
    (                                  [], [OK Clist, OK Clist, OK Clist]),
    (                             [Clist], [OK Clist, OK Clist, OK Clist]),
    (                      [Clist, Clist], [OK Clist, OK Clist, OK Clist]),
    (                        [PrimT Void], [OK Clist, OK Clist, OK Clist]),
    (                                  [], [OK Clist, OK Clist, OK Clist]),
    (                             [Clist], [OK Clist, OK Clist, OK Clist]),
    (                      [Clist, Clist], [OK Clist, OK Clist, OK Clist]),
    (                        [PrimT Void], [OK Clist, OK Clist, OK Clist])]"

lemma bounded_makelist [simp]: "check_bounded make_list_ins []"
  apply (simp add: check_bounded_def)
  apply (simp add: nat_number make_list_ins_def)
  apply (rule allI, rule impI)
  apply (elim pc_end pc_next pc_0)
  apply auto
  done
```

```
lemma types_makelist [simp]: "check_types E 3 (Suc (Suc (Suc 0))) (map OK φm)"
  apply (auto simp add: check_types_def phi_makelist_def JVM_states_unfold)
  apply (unfold list_def)
  apply auto
  done

lemma wt_makelist [simp]:
  "wt_method E test_name [] (PrimT Void) 3 2 make_list_ins [] φm"
  apply (simp add: wt_method_def)
  apply (simp add: make_list_ins_def phi_makelist_def)
  apply (simp add: wt_start_def nat_number)
  apply (simp add: wt_instr_def)
  apply clarify
  apply (elim pc_end pc_next pc_0)
  apply (simp add: match_exception_entry_def)
  apply simp
  apply simp
  apply simp
  apply (simp add: match_exception_entry_def)
  apply (simp add: match_exception_entry_def)
  apply simp
  apply simp
  apply simp
  apply (simp add: match_exception_entry_def)
  apply (simp add: match_exception_entry_def)
  apply simp
  apply simp
  apply simp
  apply (simp add: match_exception_entry_def)
  apply (simp add: match_exception_entry_def)
  apply simp
  apply (simp add: app_def xcpt_app_def)
  apply simp
  apply simp
  apply simp
  apply (simp add: app_def xcpt_app_def)
  apply simp
  done
```

The whole program is welltyped:

```
definition Phi :: prog_type ("Φ") where
  "Φ C sg ≡ if C = test_name ∧ sg = (makelist_name, []) then φm else
            if C = list_name ∧ sg = (append_name, [Class list_name]) then φa else []"

lemma wf_prog:
  "wt_jvm_prog E Φ"
  apply (unfold wt_jvm_prog_def)
  apply (rule wf_mb'E [OF wf_struct])
  apply (simp add: E_def)
  apply clarify
  apply (fold E_def)
  apply (simp add: system_defs class_defs Phi_def)
  apply auto
  done
```

4.24.4 Conformance

Execution of the program will be typesafe, because its start state conforms to the welltyping:

```
lemma "E,Φ ⊢JVM start_state E test_name makelist_name √"
  apply (rule BV_correct_initial)
    apply (rule wf_prog)
   apply simp
  apply simp
  done
```

4.24.5 Example for code generation: inferring method types

```
definition test_kil :: "jvm_prog ⇒ cname ⇒ ty list ⇒ ty ⇒ nat ⇒ nat ⇒
             exception_table ⇒ instr list ⇒ JVMType.state list" where
  "test_kil G C pTs rT mxs mxl et instr =
   (let first  = Some ([],(OK (Class C))#((map OK pTs))@(replicate mxl Err));
        start  = OK first#(replicate (size instr - 1) (OK None))
    in  kiljvm G mxs (1+size pTs+mxl) rT et instr start)"

lemma [code]:
  "unstables r step ss = (UN p:{..<size ss}. if ¬stable r step ss p then {p} else {})"
  apply (unfold unstables_def)
  apply (rule equalityI)
  apply (rule subsetI)
  apply (erule CollectE)
  apply (erule conjE)
  apply (rule UN_I)
  apply simp
  apply simp
  apply (rule subsetI)
  apply (erule UN_E)
  apply (case_tac "¬ stable r step ss p")
  apply simp+
  done

definition some_elem :: "'a set ⇒ 'a" where
  "some_elem = (%S. SOME x. x : S)"

consts_code
  "some_elem" ("(case/ _ of/ {*Set*}/ xs/ =>/ hd/ xs)")
```

This code setup is just a demonstration and *not* sound!

```
lemma False
proof -
  have "some_elem (set [False, True]) = False"
    by evaluation
  moreover have "some_elem (set [True, False]) = True"
    by evaluation
  ultimately show False
    by (simp add: some_elem_def)
qed

lemma [code]:
  "iter f step ss w = while (λ(ss, w). ¬ is_empty w)
```

```
    (λ(ss, w).
        let p = some_elem w in propa f (step p (ss ! p)) ss (w - {p}))
    (ss, w)"
  unfolding iter_def More_Set.is_empty_def some_elem_def ..

lemma JVM_sup_unfold [code]:
 "JVMType.sup S m n = lift2 (Opt.sup
       (Product.sup (Listn.sup (JType.sup S))
         (λx y. OK (map2 (lift2 (JType.sup S)) x y))))"
  apply (unfold JVMType.sup_def JVMType.sl_def Opt.esl_def Err.sl_def
         stk_esl_def reg_sl_def Product.esl_def
         Listn.sl_def upto_esl_def JType.esl_def Err.esl_def)
  by simp

lemmas [code] = JType.sup_def [unfolded exec_lub_def] JVM_le_unfold

lemmas [code_ind] = rtranclp.rtrancl_refl converse_rtranclp_into_rtranclp

code_module BV
contains
  test1 = "test_kil E list_name [Class list_name] (PrimT Void) 3 0
    [(Suc 0, 2, 8, Xcpt NullPointer)] append_ins"
  test2 = "test_kil E test_name [] (PrimT Void) 3 2 [] make_list_ins"
ML BV.test1
ML BV.test2

end
```

```
theory AuxLemmas
imports "../J/JBasis"
begin
```

```
lemma app_nth_greater_len [rule_format (no_asm), simp]:
 "∀ ind. length pre ≤ ind ⟶ (pre @ a # post) ! (Suc ind) = (pre @ post) ! ind"
apply (induct "pre")
apply auto
apply (case_tac ind)
apply auto
done

lemma length_takeWhile: "v ∈ set xs ⟹ length (takeWhile (%z. z~=v) xs) < length xs"
by (induct xs, auto)
```

```
lemma nth_length_takeWhile [simp]:
  "v ∈ set xs ⟹ xs ! (length (takeWhile (%z. z~=v) xs)) = v"
by (induct xs, auto)
```

```
lemma map_list_update [simp]:
  "⟦ x ∈ set xs; distinct xs⟧ ⟹
  (map f xs) [length (takeWhile (λz. z ≠ x) xs) := v] =
  map (f(x:=v)) xs"
apply (induct xs)
apply simp
apply (case_tac "x=a")
apply auto
done
```

```
lemma split_compose: "(split f) ∘ (λ (a,b). ((fa a), (fb b))) =
  (λ (a,b). (f (fa a) (fb b)))"
by (simp add: split_def o_def)

lemma split_iter: "(λ (a,b,c). ((g1 a), (g2 b), (g3 c))) =
        (λ (a,p). ((g1 a), (λ (b, c). ((g2 b), (g3 c))) p))"
by (simp add: split_def o_def)
```

```
lemma singleton_in_set: "A = {a} ⟹ a ∈ A" by simp
```

```
lemma the_map_upd: "(the ∘ f(x↦v)) = (the ∘ f)(x:=v)"
by (simp add: expand_fun_eq)

lemma map_of_in_set:
  "(map_of xs x = None) = (x ∉ set (map fst xs))"
by (induct xs, auto)

lemma map_map_upd [simp]:
  "y ∉ set xs ⟹ map (the ∘ f(y↦v)) xs = map (the ∘ f) xs"
by (simp add: the_map_upd)

lemma map_map_upds [rule_format (no_asm), simp]:
"∀ f vs. (∀y∈set ys. y ∉ set xs) ⟶ map (the ∘ f(ys[↦]vs)) xs = map (the ∘ f) xs"
apply (induct xs)
```

```
 apply simp
apply fastsimp
done

lemma map_upds_distinct [rule_format (no_asm), simp]:
  "∀ f vs. length ys = length vs ⟶ distinct ys ⟶ map (the ∘ f(ys[↦]vs)) ys = vs"
apply (induct ys)
apply simp
apply (intro strip)
apply (case_tac vs)
apply simp
apply (simp only: map_upds_Cons distinct.simps hd.simps tl.simps map.simps)
apply clarify
apply (simp del: o_apply)
apply simp
done


lemma map_of_map_as_map_upd [rule_format (no_asm)]: "distinct (map f zs) ⟶
map_of (map (λ p. (f p, g p)) zs) = empty (map f zs [↦] map g zs)"
by (induct zs, auto)


lemma map_upds_SomeD [rule_format (no_asm)]:
  "∀ m ys. (m(xs[↦]ys)) k = Some y ⟶ k ∈ (set xs) ∨ (m k = Some y)"
apply (induct xs)
apply simp
apply auto
apply(case_tac ys)
apply auto
done

lemma map_of_upds_SomeD: "(map_of m (xs[↦]ys)) k = Some y
  ⟹ k ∈ (set (xs @ map fst m))"
by (auto dest: map_upds_SomeD map_of_SomeD fst_in_set_lemma)


lemma map_of_map_prop [rule_format (no_asm)]:
  "(map_of (map f xs) k = Some v) ⟶
  (∀ x ∈ set xs. (P1 x)) ⟶
  (∀ x. (P1 x) ⟶ (P2 (f x))) ⟶
  (P2(k, v))"
by (induct xs,auto)

lemma map_of_map2: "∀ x ∈ set xs. (fst (f x)) = (fst x) ⟹
  map_of (map f xs) a = Option.map (λ b. (snd (f (a, b)))) (map_of xs a)"
by (induct xs, auto)

lemma option_map_of [simp]: "(Option.map f (map_of xs k) = None) = ((map_of xs k) = None)"
by (simp add: Option.map_def split: option.split)



end
```

```
theory DefsComp
imports "../JVM/JVMExec"
begin


definition method_rT :: "cname × ty × 'c ⇒ ty" where
  "method_rT mtd == (fst (snd mtd))"



definition
  gx :: "xstate ⇒ val option" where "gx ≡ fst"

definition
  gs :: "xstate ⇒ state" where "gs ≡ snd"

definition
  gh :: "xstate ⇒ aheap" where "gh ≡ fst∘snd"

definition
  gl :: "xstate ⇒ State.locals" where "gl ≡ snd∘snd"

definition
  gmb :: "'a prog ⇒ cname ⇒ sig ⇒ 'a"
    where "gmb G cn si ≡ snd(snd(the(method (G,cn) si)))"

definition
  gis :: "jvm_method ⇒ bytecode"
    where "gis ≡ fst ∘ snd ∘ snd"


definition
  gjmb_pns  :: "java_mb ⇒ vname list" where "gjmb_pns ≡ fst"

definition
  gjmb_lvs  :: "java_mb ⇒ (vname×ty)list" where  "gjmb_lvs ≡ fst∘snd"

definition
  gjmb_blk  :: "java_mb ⇒ stmt" where  "gjmb_blk ≡ fst∘snd∘snd"

definition
  gjmb_res  :: "java_mb ⇒ expr" where  "gjmb_res ≡ snd∘snd∘snd"

definition
  gjmb_plns :: "java_mb ⇒ vname list"
    where  "gjmb_plns ≡ λjmb. gjmb_pns jmb @ map fst (gjmb_lvs jmb)"

definition
  glvs :: "java_mb ⇒ State.locals ⇒ locvars"
```

```
  where "glvs jmb loc ≡ map (the∘loc) (gjmb_plns jmb)"

lemmas gdefs = gx_def gh_def gl_def gmb_def gis_def glvs_def
lemmas gjmbdefs = gjmb_pns_def gjmb_lvs_def gjmb_blk_def gjmb_res_def gjmb_plns_def

lemmas galldefs = gdefs gjmbdefs

definition locvars_locals :: "java_mb prog ⇒ cname ⇒ sig ⇒ State.locals ⇒ locvars"
where
  "locvars_locals G C S lvs == the (lvs This) # glvs (gmb G C S) lvs"

definition locals_locvars :: "java_mb prog ⇒ cname ⇒ sig ⇒ locvars ⇒ State.locals"
where
  "locals_locvars G C S lvs ==
  empty ((gjmb_plns (gmb G C S))[↦](tl lvs)) (This↦(hd lvs))"

definition locvars_xstate :: "java_mb prog ⇒ cname ⇒ sig ⇒ xstate ⇒ locvars" where
  "locvars_xstate G C S xs == locvars_locals G C S (gl xs)"


lemma locvars_xstate_par_dep:
  "lv1 = lv2 ⟹
  locvars_xstate G C S (xcpt1, hp1, lv1) = locvars_xstate G C S (xcpt2, hp2, lv2)"
by (simp add: locvars_xstate_def gl_def)


lemma gx_conv [simp]: "gx (xcpt, s) = xcpt" by (simp add: gx_def)

lemma gh_conv [simp]: "gh (xcpt, h, l) = h" by (simp add: gh_def)


end
```

```
theory Index
imports AuxLemmas DefsComp
begin


definition index :: "java_mb => vname => nat" where
 "index ==  λ (pn,lv,blk,res) v.
  if v = This
  then 0
  else Suc (length (takeWhile (λ z. z~=v) (pn @ map fst lv)))"
```

```
lemma index_length_pns: "
  ⟦ i = index (pns,lvars,blk,res) vn;
  wf_java_mdecl G C ((mn,pTs),rT, (pns,lvars,blk,res));
  vn ∈ set pns⟧
  ⟹ 0 < i ∧ i < Suc (length pns)"
apply (simp add: wf_java_mdecl_def index_def)
apply (subgoal_tac "vn ≠ This")
apply (auto intro: length_takeWhile)
done

lemma index_length_lvars: "
  ⟦ i = index (pns,lvars,blk,res) vn;
  wf_java_mdecl G C ((mn,pTs),rT, (pns,lvars,blk,res));
  vn ∈ set (map fst lvars)⟧
  ⟹ (length pns) < i ∧ i < Suc((length pns) + (length lvars))"
apply (simp add: wf_java_mdecl_def index_def)
apply (subgoal_tac "vn ≠ This")
apply simp
apply (subgoal_tac "∀ x ∈ set pns. (λz. z ≠ vn) x")
apply (simp add: takeWhile_append2)
apply (subgoal_tac "length (takeWhile (λz. z ≠ vn) (map fst lvars)) < length (map fst
lvars)")
apply simp
apply (rule length_takeWhile)
apply simp
apply (simp add: map_of_in_set)
apply (intro strip notI) apply simp apply blast
done



lemma select_at_index :
  "x ∈ set (gjmb_plns (gmb G C S)) ∨ x = This
  ⟹ (the (loc This) # glvs (gmb G C S) loc) ! (index (gmb G C S) x) =
     the (loc x)"
apply (simp only: index_def gjmb_plns_def)
apply (case_tac "gmb G C S" rule: prod.exhaust)
apply (simp add: galldefs del: set_append map_append)
apply (case_tac b, simp add: gmb_def gjmb_lvs_def del: set_append map_append)
apply (intro strip)
apply (simp del: set_append map_append)
apply (frule length_takeWhile)
apply (frule_tac f = "(the ∘ loc)" in nth_map)
apply simp
done

lemma lift_if: "(f (if b then t else e)) = (if b then (f t) else (f e))"
apply auto
done

lemma update_at_index: "
  ⟦ distinct (gjmb_plns (gmb G C S));
```

```
  x ∈ set (gjmb_plns (gmb G C S)); x ≠ This ⟧ ⟹
  locvars_xstate G C S (Norm (h, l))[index (gmb G C S) x := val] =
          locvars_xstate G C S (Norm (h, l(x↦val)))"
apply (simp only: locvars_xstate_def locvars_locals_def index_def)
apply (case_tac "gmb G C S" rule: prod.exhaust, simp)
apply (case_tac b, simp)
apply (rule conjI)
apply (simp add: gl_def)
apply (simp add: galldefs del: set_append map_append)
done
```

```
lemma index_of_var: "⟦ xvar ∉ set pns; xvar ∉ set (map fst zs); xvar ≠ This ⟧
  ⟹ index (pns, zs @ ((xvar, xval) # xys), blk, res) xvar = Suc (length pns + length
zs)"
apply (simp add: index_def)
apply (subgoal_tac "(⋀x. ((x ∈ (set pns)) ⟹ ((λz. (z ≠ xvar))x)))")
apply (simp add: List.takeWhile_append2)
apply (subgoal_tac "(takeWhile (λz. z ≠ xvar) (map fst zs @ xvar # map fst xys)) = map
fst zs @ (takeWhile (λz. z ≠ xvar) (xvar # map fst xys))")
apply simp
apply (rule List.takeWhile_append2)
apply auto
done
```

```
definition disjoint_varnames :: "[vname list, (vname × ty) list] ⇒ bool" where

  "disjoint_varnames pns lvars ≡
  distinct pns ∧ unique lvars ∧ This ∉ set pns ∧ This ∉ set (map fst lvars) ∧
  (∀pn∈set pns. pn ∉ set (map fst lvars))"
```

```
lemma index_of_var2: "
  disjoint_varnames pns (lvars_pre @ (vn, ty) # lvars_post)
  ⟹ index (pns, lvars_pre @ (vn, ty) # lvars_post, blk, res) vn =
  Suc (length pns + length lvars_pre)"
apply (simp add: disjoint_varnames_def index_def unique_def distinct_append)
apply (subgoal_tac "vn ≠ This", simp)
apply (subgoal_tac
  "takeWhile (λz. z ≠ vn) (map fst lvars_pre @ vn # map fst lvars_post) =
  map fst lvars_pre @ takeWhile (λz. z ≠ vn) (vn # map fst lvars_post)")
apply simp
apply (rule List.takeWhile_append2)
apply auto
done
```

```
lemma wf_java_mdecl_disjoint_varnames:
  "wf_java_mdecl G C (S,rT,(pns,lvars,blk,res))
```

```
  ⟹ disjoint_varnames pns lvars"
apply (case_tac S)
apply (simp add: wf_java_mdecl_def disjoint_varnames_def  map_of_in_set)
done

lemma wf_java_mdecl_length_pTs_pns:
  "wf_java_mdecl G C ((mn, pTs), rT, pns, lvars, blk, res)
  ⟹ length pTs = length pns"
by (simp add: wf_java_mdecl_def)

end
```

```
theory TranslCompTp
imports Index "../BV/JVMType"
begin
```

```
definition comb :: "['a ⇒ 'b list × 'c, 'c ⇒ 'b list × 'd, 'a] ⇒ 'b list × 'd" where

  "comb == (λ f1 f2 x0. let (xs1, x1) = f1 x0;
                            (xs2, x2) = f2 x1
                        in  (xs1 @ xs2, x2))"

definition comb_nil :: "'a ⇒ 'b list × 'a" where
  "comb_nil a == ([], a)"

notation (xsymbols)
  comb  (infixr "□" 55)

lemma comb_nil_left [simp]: "comb_nil □ f = f"
by (simp add: comb_def comb_nil_def split_beta)

lemma comb_nil_right [simp]: "f □ comb_nil = f"
by (simp add: comb_def comb_nil_def split_beta)

lemma comb_assoc [simp]: "(fa □ fb) □ fc = fa □ (fb □ fc)"
by (simp add: comb_def split_beta)

lemma comb_inv: "(xs', x') = (f1 □ f2) x0 ⟹
  ∃ xs1 x1 xs2 x2. (xs1, x1) = (f1 x0) ∧ (xs2, x2) = f2 x1 ∧ xs'= xs1 @ xs2 ∧ x'=x2"
apply (case_tac "f1 x0")
apply (case_tac "f2 x1")
apply (simp add: comb_def split_beta)
done
```

```
abbreviation (input)
  mt_of :: "method_type × state_type ⇒ method_type"
  where "mt_of == fst"
```

```
abbreviation (input)
  sttp_of :: "method_type × state_type ⇒ state_type"
  where "sttp_of == snd"

consts
  compTpExpr  :: "java_mb ⇒ java_mb prog ⇒ expr
                  ⇒ state_type ⇒ method_type × state_type"

  compTpExprs :: "java_mb ⇒ java_mb prog ⇒ expr list
                  ⇒ state_type ⇒ method_type × state_type"

  compTpStmt  :: "java_mb ⇒ java_mb prog ⇒ stmt
                  ⇒ state_type ⇒ method_type × state_type"

definition nochangeST :: "state_type ⇒ method_type × state_type" where
  "nochangeST sttp == ([Some sttp], sttp)"

definition pushST :: "[ty list, state_type] ⇒ method_type × state_type" where
  "pushST tps == (λ (ST, LT). ([Some (ST, LT)], (tps @ ST, LT)))"

definition dupST :: "state_type ⇒ method_type × state_type" where
  "dupST == (λ (ST, LT). ([Some (ST, LT)], (hd ST # ST, LT)))"

definition dup_x1ST :: "state_type ⇒ method_type × state_type" where
  "dup_x1ST == (λ (ST, LT). ([Some (ST, LT)],
                             (hd ST # hd (tl ST) # hd ST # (tl (tl ST)), LT)))"

definition popST :: "[nat, state_type] ⇒ method_type × state_type" where
  "popST n == (λ (ST, LT). ([Some (ST, LT)], (drop n ST, LT)))"

definition replST :: "[nat, ty, state_type] ⇒ method_type × state_type" where
  "replST n tp == (λ (ST, LT). ([Some (ST, LT)], (tp # (drop n ST), LT)))"

definition storeST :: "[nat, ty, state_type] ⇒ method_type × state_type" where
  "storeST i tp == (λ (ST, LT). ([Some (ST, LT)], (tl ST, LT [i:= OK tp])))"



primrec

  "compTpExpr jmb G (NewC c) = pushST [Class c]"

  "compTpExpr jmb G (Cast c e) =
  (compTpExpr jmb G e) □ (replST 1 (Class c))"

  "compTpExpr jmb G (Lit val) = pushST [the (typeof (λv. None) val)]"

  "compTpExpr jmb G (BinOp bo e1 e2) =
     (compTpExpr jmb G e1) □ (compTpExpr jmb G e2) □
     (case bo of
       Eq => popST 2 □ pushST [PrimT Boolean] □ popST 1 □ pushST [PrimT Boolean]
     | Add => replST 2 (PrimT Integer))"
```

```
"compTpExpr jmb G (LAcc vn) = (λ (ST, LT).
    pushST [ok_val (LT ! (index jmb vn))] (ST, LT))"

"compTpExpr jmb G (vn::=e) =
    (compTpExpr jmb G e) □ dupST □ (popST 1)"


"compTpExpr jmb G ( {cn}e..fn ) =
    (compTpExpr jmb G e) □ replST 1 (snd (the (field (G,cn) fn)))"

"compTpExpr jmb G (FAss cn e1 fn e2 ) =
    (compTpExpr jmb G e1) □ (compTpExpr jmb G e2) □ dup_x1ST □ (popST 2)"


"compTpExpr jmb G ({C}a..mn({fpTs}ps)) =
     (compTpExpr jmb G a) □ (compTpExprs jmb G ps) □
     (replST ((length ps) + 1) (method_rT (the (method (G,C) (mn,fpTs)))))"




"compTpExprs jmb G [] = comb_nil"

"compTpExprs jmb G (e#es) = (compTpExpr jmb G e) □ (compTpExprs jmb G es)"
```

primrec

```
 "compTpStmt jmb G Skip = comb_nil"

 "compTpStmt jmb G (Expr e) =  (compTpExpr jmb G e) □ popST 1"

 "compTpStmt jmb G (c1;; c2) = (compTpStmt jmb G c1) □ (compTpStmt jmb G c2)"

 "compTpStmt jmb G (If(e) c1 Else c2) =
    (pushST [PrimT Boolean]) □ (compTpExpr jmb G e) □ popST 2 □
    (compTpStmt jmb G c1) □ nochangeST □ (compTpStmt jmb G c2)"

 "compTpStmt jmb G (While(e) c) =
    (pushST [PrimT Boolean]) □ (compTpExpr jmb G e) □ popST 2 □
    (compTpStmt jmb G c) □ nochangeST"
```

definition *compTpInit :: "java_mb ⇒ (vname * ty)*
⇒ state_type ⇒ method_type × state_type" **where**

```
"compTpInit jmb == (λ (vn,ty). (pushST [ty]) □  (storeST (index jmb vn) ty))"
```

consts

```
compTpInitLvars :: "[java_mb, (vname × ty) list]
                  ⇒ state_type ⇒ method_type × state_type"
```

primrec

```
"compTpInitLvars jmb [] = comb_nil"
"compTpInitLvars jmb (lv#lvars) = (compTpInit jmb lv) □ (compTpInitLvars jmb lvars)"
```

```
definition start_ST :: "opstack_type" where
  "start_ST == []"

definition start_LT :: "cname ⇒ ty list ⇒ nat ⇒ locvars_type" where
  "start_LT C pTs n ==  (OK (Class C))#((map OK pTs))@(replicate n Err)"

definition compTpMethod  :: "[java_mb prog, cname, java_mb mdecl] ⇒ method_type" where
  "compTpMethod G C == λ ((mn,pTs),rT, jmb).
                         let (pns,lvars,blk,res) = jmb
                         in (mt_of
                            ((compTpInitLvars jmb lvars □
                              compTpStmt jmb G blk □
                              compTpExpr jmb G res □
                              nochangeST)
                                (start_ST, start_LT C pTs (length lvars))))"

definition compTp :: "java_mb prog => prog_type" where
  "compTp G C sig == let (D, rT, jmb) = (the (method (G, C) sig))
                      in compTpMethod G C (sig, rT, jmb)"
```

```
definition ssize_sto :: "(state_type option) ⇒ nat" where
  "ssize_sto sto ==  case sto of None ⇒ 0 | (Some (ST, LT)) ⇒ length ST"

definition max_of_list :: "nat list ⇒ nat" where
  "max_of_list xs == foldr max xs 0"

definition max_ssize :: "method_type ⇒ nat" where
  "max_ssize mt == max_of_list (map ssize_sto mt)"

end
```

```
theory TranslComp imports  TranslCompTp begin
```

```
consts
 compExpr  :: "java_mb => expr      => instr list"
 compExprs :: "java_mb => expr list => instr list"
 compStmt  :: "java_mb => stmt      => instr list"
```

primrec

```
"compExpr jmb (NewC c) = [New c]"

"compExpr jmb (Cast c e) = compExpr jmb e @ [Checkcast c]"

"compExpr jmb (Lit val) = [LitPush val]"

"compExpr jmb (BinOp bo e1 e2) = compExpr jmb e1 @ compExpr jmb e2 @
  (case bo of Eq => [Ifcmpeq 3,LitPush(Bool False),Goto 2,LitPush(Bool True)]
            | Add => [IAdd])"

"compExpr jmb (LAcc vn) = [Load (index jmb vn)]"

"compExpr jmb (vn::=e) = compExpr jmb e @ [Dup , Store (index jmb vn)]"

"compExpr jmb ( {cn}e..fn ) = compExpr jmb e @ [Getfield fn cn]"

"compExpr jmb (FAss cn e1 fn e2 ) =
       compExpr jmb e1 @ compExpr jmb e2 @ [Dup_x1 , Putfield fn cn]"

"compExpr jmb (Call cn e1 mn X ps) =
        compExpr jmb e1 @ compExprs jmb ps @ [Invoke cn mn X]"

"compExprs jmb []     = []"

"compExprs jmb (e#es) = compExpr jmb e @ compExprs jmb es"
```

primrec

```
"compStmt jmb Skip = []"

"compStmt jmb (Expr e) = ((compExpr jmb e) @ [Pop])"
```

```
"compStmt jmb (c1;; c2) = ((compStmt jmb c1) @ (compStmt jmb c2))"

"compStmt jmb (If(e) c1 Else c2) =
        (let cnstf = LitPush (Bool False);
             cnd   = compExpr jmb e;
             thn   = compStmt jmb c1;
             els   = compStmt jmb c2;
             test  = Ifcmpeq (int(length thn +2));
             thnex = Goto (int(length els +1))
         in
         [cnstf] @ cnd @ [test] @ thn @ [thnex] @ els)"

"compStmt jmb (While(e) c) =
        (let cnstf = LitPush (Bool False);
             cnd   = compExpr jmb e;
             bdy   = compStmt jmb c;
             test  = Ifcmpeq (int(length bdy +2));
             loop  = Goto (-(int((length bdy) + (length cnd) +2)))
         in
         [cnstf] @ cnd @ [test] @ bdy @ [loop])"
```

definition *load_default_val :: "ty => instr"* **where**
"load_default_val ty == LitPush (default_val ty)"

definition *compInit :: "java_mb => (vname * ty) => instr list"* **where**
"compInit jmb == λ (vn,ty). [load_default_val ty, Store (index jmb vn)]"

definition *compInitLvars :: "[java_mb, (vname × ty) list] ⇒ bytecode"* **where**
"compInitLvars jmb lvars == concat (map (compInit jmb) lvars)"

definition *compMethod :: "java_mb prog ⇒ cname ⇒ java_mb mdecl ⇒ jvm_method mdecl"* **where**

```
  "compMethod G C jmdl == let (sig, rT, jmb) = jmdl;
                        (pns,lvars,blk,res) = jmb;
                        mt = (compTpMethod G C jmdl);
                        bc = compInitLvars jmb lvars @
                             compStmt jmb blk @ compExpr jmb res @
                             [Return]
                  in (sig, rT, max_ssize mt, length lvars, bc, [])"
```

definition *compClass :: "java_mb prog => java_mb cdecl=> jvm_method cdecl"* **where**
"compClass G == λ (C,cno,fdls,jmdls). (C,cno,fdls, map (compMethod G C) jmdls)"

definition *comp :: "java_mb prog => jvm_prog"* **where**
"comp G == map (compClass G) G"

end

```
theory LemmasComp
imports TranslComp
begin


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


lemma split_pairs: "(λ(a,b). (F a b)) (ab) = F (fst ab) (snd ab)"
proof -
  have "(λ(a,b). (F a b)) (fst ab,snd ab) = F (fst ab) (snd ab)"
    by (simp add: split_def)
  then show ?thesis by simp
qed


lemma c_hupd_conv:
  "c_hupd h' (xo, (h,l)) = (xo, (if xo = None then h' else h),l)"
by (simp add: c_hupd_def)

lemma gl_c_hupd [simp]: "(gl (c_hupd h xs)) = (gl xs)"
by (simp add: gl_def c_hupd_def split_pairs)

lemma c_hupd_xcpt_invariant [simp]: "gx (c_hupd h' (xo, st)) = xo"
by (case_tac st, simp only: c_hupd_conv gx_conv)


lemma c_hupd_hp_invariant: "gh (c_hupd hp (None, st)) = hp"
by (case_tac st, simp add: c_hupd_conv gh_def)


lemma unique_map_fst [rule_format]: "(∀ x ∈ set xs. (fst x = fst (f x) )) ⟶
  unique (map f xs) = unique xs"
proof (induct xs)
  case Nil show ?case by simp
next
  case (Cons a list)
  show ?case
  proof
    assume fst_eq: "∀x∈set (a # list). fst x = fst (f x)"

    have fst_set: "(fst a ∈ fst ` set list) = (fst (f a) ∈ fst ` f ` set list)"
    proof
      assume as: "fst a ∈ fst ` set list"
```

```
      then obtain x where fst_a_x: "x∈set list ∧ fst a = fst x"
        by (auto simp add: image_iff)
      then have "fst (f a) = fst (f x)" by (simp add: fst_eq)
      with as show "(fst (f a) ∈ fst ‘ f ‘ set list)" by (simp add: fst_a_x)
    next
      assume as: "fst (f a) ∈ fst ‘ f ‘ set list"
      then obtain x where fst_a_x: "x∈set list ∧ fst (f a) = fst (f x)"
        by (auto simp add: image_iff)
      then have "fst a = fst x" by (simp add: fst_eq)
      with as show "fst a ∈ fst ‘ set list" by (simp add: fst_a_x)
    qed

    with fst_eq Cons
    show "unique (map f (a # list)) = unique (a # list)"
      by (simp add: unique_def fst_set image_compose)
  qed
qed

lemma comp_unique: "unique (comp G) = unique G"
apply (simp add: comp_def)
apply (rule unique_map_fst)
apply (simp add: compClass_def split_beta)
done




lemma comp_class_imp:
  "(class G C = Some(D, fs, ms)) ⟹
  (class (comp G) C = Some(D, fs, map (compMethod G C) ms))"
apply (simp add: class_def comp_def compClass_def)
apply (rule HOL.trans)
apply (rule map_of_map2)
apply auto
done

lemma comp_class_None:
"(class G C = None) = (class (comp G) C = None)"
apply (simp add: class_def comp_def compClass_def)
apply (simp add: map_of_in_set)
apply (simp add: image_compose [THEN sym] o_def split_beta del: image_compose)
done

lemma comp_is_class: "is_class (comp G) C = is_class G C"
by (case_tac "class G C", auto simp: is_class_def comp_class_None dest: comp_class_imp)


lemma comp_is_type: "is_type (comp G) T = is_type G T"
  by ((cases T),simp,(induct G)) ((simp),(simp only: comp_is_class),(simp add: comp_is_class),(simp
only: comp_is_class))

lemma comp_classname: "is_class G C
  ⟹ fst (the (class G C)) = fst (the (class (comp G) C))"
```

```
by (case_tac "class G C", auto simp: is_class_def dest: comp_class_imp)

lemma comp_subcls1: "subcls1 (comp G) = subcls1 G"
by (auto simp add: subcls1_def2 comp_classname comp_is_class)

lemma comp_widen: "widen (comp G) = widen G"
  apply (simp add: expand_fun_eq)
  apply (intro allI iffI)
  apply (erule widen.cases)
  apply (simp_all add: comp_subcls1 widen.null)
  apply (erule widen.cases)
  apply (simp_all add: comp_subcls1 widen.null)
  done

lemma comp_cast: "cast (comp G) = cast G"
  apply (simp add: expand_fun_eq)
  apply (intro allI iffI)
  apply (erule cast.cases)
  apply (simp_all add: comp_subcls1 cast.widen cast.subcls)
  apply (rule cast.widen) apply (simp add: comp_widen)
  apply (erule cast.cases)
  apply (simp_all add: comp_subcls1 cast.widen cast.subcls)
  apply (rule cast.widen) apply (simp add: comp_widen)
  done

lemma comp_cast_ok: "cast_ok (comp G) = cast_ok G"
  by (simp add: expand_fun_eq cast_ok_def comp_widen)


lemma compClass_fst [simp]: "(fst (compClass G C)) = (fst C)"
by (simp add: compClass_def split_beta)

lemma compClass_fst_snd [simp]: "(fst (snd (compClass G C))) = (fst (snd C))"
by (simp add: compClass_def split_beta)

lemma compClass_fst_snd_snd [simp]: "(fst (snd (snd (compClass G C)))) = (fst (snd (snd
C)))"
by (simp add: compClass_def split_beta)

lemma comp_wf_fdecl [simp]: "wf_fdecl (comp G) fd = wf_fdecl G fd"
by (case_tac fd, simp add: wf_fdecl_def comp_is_type)


lemma compClass_forall [simp]: "
  (∀x∈set (snd (snd (snd (compClass G C)))). P (fst x) (fst (snd x))) =
  (∀x∈set (snd (snd (snd C))). P (fst x) (fst (snd x)))"
by (simp add: compClass_def compMethod_def split_beta)

lemma comp_wf_mhead: "wf_mhead (comp G) S rT =  wf_mhead G S rT"
by (simp add: wf_mhead_def split_beta comp_is_type)

lemma comp_ws_cdecl: "
  ws_cdecl (TranslComp.comp G) (compClass G C) = ws_cdecl G C"
apply (simp add: ws_cdecl_def split_beta comp_is_class comp_subcls1)
```

```
apply (simp (no_asm_simp) add: comp_wf_mhead)
apply (simp add: compClass_def compMethod_def split_beta unique_map_fst)
done


lemma comp_wf_syscls: "wf_syscls (comp G) = wf_syscls G"
apply (simp add: wf_syscls_def comp_def compClass_def split_beta)
apply (simp only: image_compose [THEN sym])
apply (subgoal_tac "(Fun.comp fst (λ(C, cno::cname, fdls::fdecl list, jmdls).
  (C, cno, fdls, map (compMethod G C) jmdls))) = fst")
apply (simp del: image_compose)
apply (simp add: expand_fun_eq split_beta)
done


lemma comp_ws_prog: "ws_prog (comp G) = ws_prog G"
apply (rule sym)
apply (simp add: ws_prog_def comp_ws_cdecl comp_unique)
apply (simp add: comp_wf_syscls)
apply (auto simp add: comp_ws_cdecl [THEN sym] TranslComp.comp_def)
done


lemma comp_class_rec: " wf ((subcls1 G)^-1) ⟹
class_rec (comp G) C t f =
  class_rec G C t (λ C' fs' ms' r'. f C' fs' (map (compMethod G C') ms') r')"
apply (rule_tac a = C in  wf_induct) apply assumption
apply (subgoal_tac "wf ((subcls1 (comp G))^-1)")
apply (subgoal_tac "(class G x = None) ∨ (∃ D fs ms. (class G x = Some (D, fs, ms)))")
apply (erule disjE)


apply (simp (no_asm_simp) add: class_rec_def comp_subcls1
  wfrec [where r="(subcls1 G)^-1", simplified])
apply (simp add: comp_class_None)


apply (erule exE)+
apply (frule comp_class_imp)
apply (frule_tac G="comp G" and C=x and t=t and f=f in class_rec_lemma)
  apply assumption
apply (frule_tac G=G and C=x and t=t
   and f="(λC' fs' ms'. f C' fs' (map (compMethod G C') ms'))" in class_rec_lemma)
  apply assumption
apply (simp only:)

apply (case_tac "x = Object")
  apply simp
  apply (frule subcls1I, assumption)
    apply (drule_tac x=D in spec, drule mp, simp)
    apply simp


apply (case_tac "class G x")
```

```
apply auto
apply (simp add: comp_subcls1)
done

lemma comp_fields: "wf ((subcls1 G)^-1) ⟹
  fields (comp G,C) = fields (G,C)"
by (simp add: fields_def comp_class_rec)

lemma comp_field: "wf ((subcls1 G)^-1) ⟹
  field (comp G,C) = field (G,C)"
by (simp add: TypeRel.field_def comp_fields)



lemma class_rec_relation [rule_format (no_asm)]: "⟦  ws_prog G;
  ∀ fs ms. R (f1 Object fs ms t1) (f2 Object fs ms t2);
   ∀ C fs ms r1 r2. (R r1 r2) ⟶ (R (f1 C fs ms r1) (f2 C fs ms r2)) ⟧
  ⟹ ((class G C) ≠ None) ⟶
  R (class_rec G C t1 f1) (class_rec G C t2 f2)"
apply (frule wf_subcls1)
apply (rule_tac a = C in  wf_induct) apply assumption
apply (intro strip)
apply (subgoal_tac "(∃D rT mb. class G x = Some (D, rT, mb))")
  apply (erule exE)+
apply (frule_tac C=x and t=t1 and f=f1 in class_rec_lemma)
  apply assumption
apply (frule_tac C=x and t=t2 and f=f2 in class_rec_lemma)
  apply assumption
apply (simp only:)

apply (case_tac "x = Object")
  apply simp
  apply (frule subcls1I, assumption)
    apply (drule_tac x=D in spec, drule mp, simp)
    apply simp
    apply (subgoal_tac "(∃D' rT' mb'. class G D = Some (D', rT', mb'))")
    apply blast



apply (frule class_wf_struct) apply assumption
apply (simp add: ws_cdecl_def is_class_def)

apply (simp add: subcls1_def2 is_class_def)
apply auto
done


abbreviation (input)
  "mtd_mb == snd o snd"

lemma map_of_map:
  "map_of (map (λ(k, v). (k, f v)) xs) k = Option.map f (map_of xs k)"
  by (simp add: map_of_map)
```

```
lemma map_of_map_fst: "⟦ inj f;
  ∀x∈set xs. fst (f x) = fst x; ∀x∈set xs. fst (g x) = fst x ⟧
  ⟹  map_of (map g xs) k
  = Option.map (λ e. (snd (g ((inv f) (k, e))))) (map_of (map f xs) k)"
apply (induct xs)
apply simp
apply (simp del: split_paired_All)
apply (case_tac "k = fst a")
apply (simp del: split_paired_All)
apply (subgoal_tac "(inv f (fst a, snd (f a))) = a", simp)
apply (subgoal_tac "(fst a, snd (f a)) = f a", simp)
apply (erule conjE)+
apply (drule_tac s ="fst (f a)" and t="fst a" in sym)
apply simp
apply (simp add: surjective_pairing)
done

lemma comp_method [rule_format (no_asm)]: "⟦ ws_prog G; is_class G C⟧ ⟹
  ((method (comp G, C) S) =
  Option.map (λ (D,rT,b).  (D, rT, mtd_mb (compMethod G D (S, rT, b))))
            (method (G, C) S))"
apply (simp add: method_def)
apply (frule wf_subcls1)
apply (simp add: comp_class_rec)
apply (simp add: split_iter split_compose map_map [symmetric] del: map_map)
apply (rule_tac R="%x y. ((x S) = (Option.map (λ(D, rT, b).
  (D, rT, snd (snd (compMethod G D (S, rT, b))))) (y S)))"
  in class_rec_relation)
apply assumption

apply (intro strip)
apply simp

apply (rule trans)

apply (rule_tac f="(λ(s, m). (s, Object, m))" and
  g="(Fun.comp (λ(s, m). (s, Object, m)) (compMethod G Object))"
  in map_of_map_fst)
defer
apply (simp add: inj_on_def split_beta)
apply (simp add: split_beta compMethod_def)
apply (simp add: map_of_map [symmetric])
apply (simp add: split_beta)
apply (simp add: Fun.comp_def split_beta)
apply (subgoal_tac "(λx::(vname list × fdecl list × stmt × expr) mdecl.
                    (fst x, Object,
                     snd (compMethod G Object
                          (inv (λ(s::sig, m::ty × vname list × fdecl list × stmt × expr).
                                   (s, Object, m))
                            (S, Object, snd x)))))
  = (λx. (fst x, Object, fst (snd x),
                        snd (snd (compMethod G Object (S, snd x)))))")
apply (simp only:)
```

```
apply (simp add: expand_fun_eq)
apply (intro strip)
apply (subgoal_tac "(inv (λ(s, m). (s, Object, m)) (S, Object, snd x)) = (S, snd x)")
apply (simp only:)
apply (simp add: compMethod_def split_beta)
apply (rule inv_f_eq)
defer
defer

apply (intro strip)
apply (simp add: map_add_Some_iff map_of_map del: split_paired_Ex)
apply (simp add: map_add_def)
apply (subgoal_tac "inj (λ(s, m). (s, Ca, m))")
apply (drule_tac g="(Fun.comp (λ(s, m). (s, Ca, m)) (compMethod G Ca))" and xs=ms
  and k=S in map_of_map_fst)
apply (simp add: split_beta)
apply (simp add: compMethod_def split_beta)
apply (case_tac "(map_of (map (λ(s, m). (s, Ca, m)) ms) S)")
apply simp
apply simp apply (simp add: split_beta map_of_map) apply (erule exE) apply (erule conjE)+
apply (drule_tac t=a in sym)
apply (subgoal_tac "(inv (λ(s, m). (s, Ca, m)) (S, a)) = (S, snd a)")
apply simp
apply (subgoal_tac "∀x∈set ms. fst ((Fun.comp (λ(s, m). (s, Ca, m)) (compMethod G Ca))
x) = fst x")
   prefer 2 apply (simp (no_asm_simp) add: compMethod_def split_beta)
apply (simp add: map_of_map2)
apply (simp (no_asm_simp) add: compMethod_def split_beta)

  — remaining subgoals
apply (auto intro: inv_f_eq simp add: inj_on_def is_class_def)
done


lemma comp_wf_mrT: "⟦ ws_prog G; is_class G D⟧ ⟹
  wf_mrT (TranslComp.comp G) (C, D, fs, map (compMethod G a) ms) =
  wf_mrT G (C, D, fs, ms)"
apply (simp add: wf_mrT_def compMethod_def split_beta)
apply (simp add: comp_widen)
apply (rule iffI)
apply (intro strip)
apply simp
apply (drule bspec) apply assumption
apply (drule_tac x=D' in spec) apply (drule_tac x=rT' in spec) apply (drule mp)
prefer 2 apply assumption
apply (simp add: comp_method [of G D])
apply (erule exE)+
apply (subgoal_tac "z =(fst z, fst (snd z), snd (snd z))")
apply (rule exI)
apply (simp)
apply (simp add: split_paired_all)
apply (intro strip)
apply (simp add: comp_method)
```

```
apply auto
done




lemma max_spec_preserves_length:
  "max_spec G C (mn, pTs) = {((md,rT),pTs')}
  ⟹ length pTs = length pTs'"
apply (frule max_spec2mheads)
apply (erule exE)+
apply (simp add: list_all2_def)
done


lemma ty_exprs_length [simp]: "(E⊢es[::]Ts ⟶ length es = length Ts)"
apply (subgoal_tac "(E⊢e::T ⟶ True) ∧ (E⊢es[::]Ts ⟶ length es = length Ts) ∧ (E⊢s√
⟶ True)")
apply blast
apply (rule ty_expr_ty_exprs_wt_stmt.induct)
apply auto
done


lemma max_spec_preserves_method_rT [simp]:
  "max_spec G C (mn, pTs) = {((md,rT),pTs')}
  ⟹ method_rT (the (method (G, C) (mn, pTs'))) = rT"
apply (frule max_spec2mheads)
apply (erule exE)+
apply (simp add: method_rT_def)
done



declare compClass_fst [simp del]
declare compClass_fst_snd [simp del]
declare compClass_fst_snd_snd [simp del]

declare split_paired_All [simp add]
declare split_paired_Ex [simp add]

end




theory CorrComp
imports "../J/JTypeSafe" "LemmasComp"
begin

declare wf_prog_ws_prog [simp add]
```

```
lemma eval_evals_exec_xcpt:
 "(G ⊢ xs -ex≻val-> xs' ⟶ gx xs' = None ⟶ gx xs = None) ∧
  (G ⊢ xs -exs[≻]vals-> xs' ⟶ gx xs' = None ⟶ gx xs = None) ∧
  (G ⊢ xs -st-> xs' ⟶ gx xs' = None ⟶ gx xs = None)"
by (induct rule: eval_evals_exec.induct, auto)


lemma eval_xcpt: "G ⊢ xs -ex≻val-> xs' ⟹ gx xs' = None ⟹ gx xs = None"
 (is "?H1 ⟹ ?H2 ⟹ ?T")
proof-
  assume h1: ?H1
  assume h2: ?H2
  from h1 h2 eval_evals_exec_xcpt show "?T" by simp
qed


lemma evals_xcpt: "G ⊢ xs -exs[≻]vals-> xs' ⟹ gx xs' = None ⟹ gx xs = None"
 (is "?H1 ⟹ ?H2 ⟹ ?T")
proof-
  assume h1: ?H1
  assume h2: ?H2
  from h1 h2 eval_evals_exec_xcpt show "?T" by simp
qed


lemma exec_xcpt: "G ⊢ xs -st-> xs' ⟹ gx xs' = None ⟹ gx xs = None"
 (is "?H1 ⟹ ?H2 ⟹ ?T")
proof-
  assume h1: ?H1
  assume h2: ?H2
  from h1 h2 eval_evals_exec_xcpt show "?T" by simp
qed


theorem exec_all_trans: "⟦(exec_all G s0 s1); (exec_all G s1 s2)⟧ ⟹ (exec_all G s0
s2)"
apply (auto simp: exec_all_def elim: Transitive_Closure.rtrancl_trans)
done

theorem exec_all_refl: "exec_all G s s"
by (simp only: exec_all_def)


theorem exec_instr_in_exec_all:
  "⟦ exec_instr i G hp stk lvars C S pc frs =  (None, hp', frs');
            gis (gmb G C S) ! pc = i⟧  ⟹
      G ⊢ (None, hp, (stk, lvars, C, S, pc) # frs) -jvm→ (None, hp', frs')"
apply (simp only: exec_all_def)
apply (rule rtrancl_refl [THEN rtrancl_into_rtrancl])
apply (simp add: gis_def gmb_def)
```

```
apply (case_tac frs', simp+)
done

theorem exec_all_one_step: "
  ⟦ gis (gmb G C S) = pre @ (i # post); pc0 = length pre;
  (exec_instr i G hp0 stk0 lvars0 C S pc0 frs) =
  (None, hp1, (stk1,lvars1,C,S, Suc pc0)#frs) ⟧
  ⟹
  G ⊢ (None, hp0, (stk0,lvars0,C,S, pc0)#frs) -jvm→
  (None, hp1, (stk1,lvars1,C,S, Suc pc0)#frs)"
apply (unfold exec_all_def)
apply (rule r_into_rtrancl)
apply (simp add: gis_def gmb_def split_beta)
done




definition progression :: "jvm_prog ⇒ cname ⇒ sig ⇒
                aheap ⇒ opstack ⇒ locvars ⇒
                bytecode ⇒
                aheap ⇒ opstack ⇒ locvars ⇒
                bool"
  ("{_,_,_} ⊢ {_, _, _} >- _ → {_, _, _}" [61,61,61,61,61,61,90,61,61,61]60) where
  "{G,C,S} ⊢ {hp0, os0, lvars0} >- instrs → {hp1, os1, lvars1} ==
  ∀ pre post frs.
  (gis (gmb G C S) = pre @ instrs @ post) ⟶
   G ⊢ (None,hp0,(os0,lvars0,C,S,length pre)#frs) -jvm→
       (None,hp1,(os1,lvars1,C,S,(length pre) + (length instrs))#frs)"



lemma progression_call:
  "⟦ ∀ pc frs.
  exec_instr instr G hp0 os0 lvars0 C S pc frs =
      (None, hp', (os', lvars', C', S', 0) # (fr pc) # frs) ∧
  gis (gmb G C' S') = instrs' @ [Return] ∧
  {G, C', S'} ⊢ {hp', os', lvars'} >- instrs' → {hp'', os'', lvars''}  ∧
  exec_instr Return G hp'' os'' lvars'' C' S' (length instrs')
                                              ((fr pc) # frs) =
      (None, hp2, (os2, lvars2, C, S, Suc pc) # frs) ⟧ ⟹
  {G, C, S} ⊢ {hp0, os0, lvars0} >-[instr]→ {hp2,os2,lvars2}"
apply (simp only: progression_def)
apply (intro strip)
apply (drule_tac x="length pre" in spec)
apply (drule_tac x="frs" in spec)
apply clarify
apply (rule exec_all_trans)
apply (rule exec_instr_in_exec_all) apply simp
apply simp
apply (rule exec_all_trans)
apply (simp only: append_Nil)
apply (drule_tac x="[]" in spec)
apply (simp only: list.simps list.size)
```

```
apply blast
apply (rule exec_instr_in_exec_all)
apply auto
done


lemma progression_transitive:
  "⟦ instrs_comb = instrs0 @ instrs1;
  {G, C, S} ⊢ {hp0, os0, lvars0} >- instrs0 → {hp1, os1, lvars1};
  {G, C, S} ⊢ {hp1, os1, lvars1} >- instrs1 → {hp2, os2, lvars2} ⟧
  ⟹
  {G, C, S} ⊢ {hp0, os0, lvars0} >- instrs_comb → {hp2, os2, lvars2}"
apply (simp only: progression_def)
apply (intro strip)
apply (rule_tac ?s1.0 = "Norm (hp1, (os1, lvars1, C, S,
                         length pre + length instrs0) # frs)"
       in exec_all_trans)
apply (simp only: append_assoc)
apply (erule thin_rl, erule thin_rl)
apply (drule_tac x="pre @ instrs0" in spec)
apply (simp add: add_assoc)
done

lemma progression_refl:
  "{G, C, S} ⊢ {hp0, os0, lvars0} >- [] → {hp0, os0, lvars0}"
apply (simp add: progression_def)
apply (intro strip)
apply (rule exec_all_refl)
done

lemma progression_one_step: "
  ∀ pc frs.
  (exec_instr i G hp0 os0 lvars0 C S pc frs) =
  (None, hp1, (os1,lvars1,C,S, Suc pc)#frs)
  ⟹ {G, C, S} ⊢ {hp0, os0, lvars0} >- [i] → {hp1, os1, lvars1}"
apply (unfold progression_def)
apply (intro strip)
apply simp
apply (rule exec_all_one_step)
apply auto
done


definition jump_fwd :: "jvm_prog ⇒ cname ⇒ sig ⇒
                 aheap ⇒ locvars ⇒ opstack ⇒ opstack ⇒
                 instr ⇒ bytecode ⇒ bool" where
  "jump_fwd G C S hp lvars os0 os1 instr instrs ==
  ∀ pre post frs.
  (gis (gmb G C S) = pre @ instr # instrs @ post) ⟶
   exec_all G (None,hp,(os0,lvars,C,S, length pre)#frs)
    (None,hp,(os1,lvars,C,S,(length pre) + (length instrs) + 1)#frs)"


lemma jump_fwd_one_step:
```

```
  "∀ pc frs.
  exec_instr instr G hp os0 lvars C S pc frs =
    (None, hp, (os1, lvars, C, S, pc + (length instrs) + 1)#frs)
  ⟹ jump_fwd G C S hp lvars os0 os1 instr instrs"
apply (unfold jump_fwd_def)
apply (intro strip)
apply (rule exec_instr_in_exec_all)
apply auto
done


lemma jump_fwd_progression_aux:
  "⟦ instrs_comb = instr # instrs0 @ instrs1;
     jump_fwd G C S hp lvars os0 os1 instr instrs0;
     {G, C, S} ⊢ {hp, os1, lvars} >- instrs1 → {hp2, os2, lvars2} ⟧
  ⟹ {G, C, S} ⊢ {hp, os0, lvars} >- instrs_comb → {hp2, os2, lvars2}"
apply (simp only: progression_def jump_fwd_def)
apply (intro strip)
apply (rule_tac ?s1.0 = "Norm(hp, (os1, lvars, C, S, length pre + length instrs0 + 1)
# frs)" in exec_all_trans)
apply (simp only: append_assoc)
apply (subgoal_tac "pre @ (instr # instrs0 @ instrs1) @ post = pre @ instr # instrs0 @
(instrs1 @ post)")
apply blast
apply simp
apply (erule thin_rl, erule thin_rl)
apply (drule_tac x="pre @ instr # instrs0" in spec)
apply (simp add: add_assoc)
done


lemma jump_fwd_progression:
  "⟦ instrs_comb = instr # instrs0 @ instrs1;
  ∀ pc frs.
  exec_instr instr G hp os0 lvars C S pc frs =
    (None, hp, (os1, lvars, C, S, pc + (length instrs0) + 1)#frs);
  {G, C, S} ⊢ {hp, os1, lvars} >- instrs1 → {hp2, os2, lvars2} ⟧
  ⟹ {G, C, S}  ⊢ {hp, os0, lvars} >- instrs_comb → {hp2, os2, lvars2}"
apply (rule jump_fwd_progression_aux)
apply assumption
apply (rule jump_fwd_one_step) apply assumption+
done



definition jump_bwd :: "jvm_prog ⇒ cname ⇒ sig ⇒
                 aheap ⇒ locvars ⇒ opstack ⇒ opstack ⇒
                 bytecode ⇒ instr ⇒ bool" where
  "jump_bwd G C S hp lvars os0 os1 instrs instr ==
  ∀ pre post frs.
  (gis (gmb G C S) = pre @ instrs @ instr # post) ⟶
   exec_all G (None,hp,(os0,lvars,C,S, (length pre) + (length instrs))#frs)
    (None,hp,(os1,lvars,C,S, (length pre))#frs)"
```

```
lemma jump_bwd_one_step:
  "∀ pc frs.
  exec_instr instr G hp os0 lvars C S (pc + (length instrs)) frs =
    (None, hp, (os1, lvars, C, S, pc)#frs)
  ⟹
  jump_bwd G C S hp lvars os0 os1 instrs instr"
apply (unfold jump_bwd_def)
apply (intro strip)
apply (rule exec_instr_in_exec_all)
apply auto
done

lemma jump_bwd_progression:
  "⟦ instrs_comb = instrs @ [instr];
  {G, C, S} ⊢ {hp0, os0, lvars0} >- instrs → {hp1, os1, lvars1};
  jump_bwd G C S hp1 lvars1 os1 os2 instrs instr;
  {G, C, S} ⊢ {hp1, os2, lvars1} >- instrs_comb → {hp3, os3, lvars3} ⟧
  ⟹ {G, C, S}  ⊢ {hp0, os0, lvars0} >- instrs_comb → {hp3, os3, lvars3}"
apply (simp only: progression_def jump_bwd_def)
apply (intro strip)
apply (rule exec_all_trans, force)
apply (rule exec_all_trans, force)
apply (rule exec_all_trans, force)
apply simp
apply (rule exec_all_refl)
done
```

```
definition class_sig_defined :: "'c prog ⇒ cname ⇒ sig ⇒ bool" where
  "class_sig_defined G C S ==
  is_class G C ∧ (∃ D rT mb. (method (G, C) S = Some (D, rT, mb)))"
```

```
definition env_of_jmb :: "java_mb prog ⇒ cname ⇒ sig ⇒ java_mb env" where
  "env_of_jmb G C S ==
  (let (mn,pTs) = S;
       (D,rT,(pns,lvars,blk,res)) = the(method (G, C) S) in
  (G,map_of lvars(pns[↦]pTs)(This↦Class C)))"

lemma env_of_jmb_fst [simp]: "fst (env_of_jmb G C S) = G"
by (simp add: env_of_jmb_def split_beta)
```

```
lemma method_preserves [rule_format (no_asm)]:
  "⟦ wf_prog wf_mb G; is_class G C;
  ∀ S rT mb. ∀ cn ∈ fst ` set G. wf_mdecl wf_mb G cn (S,rT,mb)  ⟶ (P cn S (rT,mb))⟧
```

```
 ⟹ ∀ D.
  method (G, C) S = Some (D, rT, mb) ⟶ (P D S (rT,mb))"

apply (frule wf_prog_ws_prog [THEN wf_subcls1])
apply (rule subcls1_induct, assumption, assumption)

apply (intro strip)
apply ((drule spec)+, drule_tac x="Object" in bspec)
apply (simp add: wf_prog_def ws_prog_def wf_syscls_def)
apply (subgoal_tac "D=Object") apply simp
apply (drule mp)
apply (frule_tac C=Object in method_wf_mdecl)
 apply simp
 apply assumption apply simp apply assumption apply simp

apply (simplesubst method_rec) apply simp
apply force
apply simp
apply (simp only: map_add_def)
apply (split option.split)
apply (rule conjI)
apply force
apply (intro strip)
apply (frule_tac
  ?P1.0 = "wf_mdecl wf_mb G Ca" and
  ?P2.0 = "%(S, (Da, rT, mb)). P Da S (rT, mb)" in map_of_map_prop)
apply (force simp: wf_cdecl_def)

apply clarify

apply (simp only: class_def)
apply (drule map_of_SomeD)+
apply (frule_tac A="set G" and f=fst in imageI, simp)
apply blast
apply simp
done


lemma method_preserves_length:
  "⟦ wf_java_prog G; is_class G C;
  method (G, C) (mn,pTs) = Some (D, rT, pns, lvars, blk, res)⟧
 ⟹ length pns = length pTs"
apply (frule_tac
  P="%D (mn,pTs) (rT, pns, lvars, blk, res). length pns = length pTs"
  in method_preserves)
apply (auto simp: wf_mdecl_def wf_java_mdecl_def)
done



definition wtpd_expr :: "java_mb env ⇒ expr ⇒ bool" where
  "wtpd_expr E e == (∃ T. E⊢e :: T)"

definition wtpd_exprs :: "java_mb env ⇒ (expr list) ⇒ bool" where
```

```
  "wtpd_exprs E e == (∃ T. E⊢e [::] T)"

definition wtpd_stmt :: "java_mb env ⇒ stmt ⇒ bool" where
  "wtpd_stmt E c == (E⊢c √)"

lemma wtpd_expr_newc: "wtpd_expr E (NewC C) ⟹ is_class (prg E) C"
by (simp only: wtpd_expr_def, erule exE, erule ty_expr.cases, auto)

lemma wtpd_expr_cast: "wtpd_expr E (Cast cn e) ⟹ (wtpd_expr E e)"
by (simp only: wtpd_expr_def, erule exE, erule ty_expr.cases, auto)

lemma wtpd_expr_lacc: "⟦ wtpd_expr (env_of_jmb G C S) (LAcc vn);
  class_sig_defined G C S ⟧
  ⟹ vn ∈ set (gjmb_plns (gmb G C S)) ∨ vn = This"
apply (simp only: wtpd_expr_def env_of_jmb_def class_sig_defined_def galldefs)
apply clarify
apply (case_tac S)
apply simp
apply (erule ty_expr.cases)
apply (auto dest: map_upds_SomeD map_of_SomeD fst_in_set_lemma)
apply (drule map_upds_SomeD)
apply (erule disjE)
  apply assumption
  apply (drule map_of_SomeD) apply (auto dest: fst_in_set_lemma)
done

lemma wtpd_expr_lass: "wtpd_expr E (vn::=e)
  ⟹ (vn ≠ This) & (wtpd_expr E (LAcc vn)) & (wtpd_expr E e)"
by (simp only: wtpd_expr_def, erule exE, erule ty_expr.cases, auto)

lemma wtpd_expr_facc: "wtpd_expr E ({fd}a..fn)
  ⟹ (wtpd_expr E a)"
by (simp only: wtpd_expr_def, erule exE, erule ty_expr.cases, auto)

lemma wtpd_expr_fass: "wtpd_expr E ({fd}a..fn:=v)
  ⟹ (wtpd_expr E ({fd}a..fn)) & (wtpd_expr E v)"
by (simp only: wtpd_expr_def, erule exE, erule ty_expr.cases, auto)


lemma wtpd_expr_binop: "wtpd_expr E (BinOp bop e1 e2)
  ⟹ (wtpd_expr E e1) & (wtpd_expr E e2)"
by (simp only: wtpd_expr_def, erule exE, erule ty_expr.cases, auto)

lemma wtpd_exprs_cons: "wtpd_exprs E (e # es)
  ⟹ (wtpd_expr E e) & (wtpd_exprs E es)"
by (simp only: wtpd_exprs_def wtpd_expr_def, erule exE, erule ty_exprs.cases, auto)

lemma wtpd_stmt_expr: "wtpd_stmt E (Expr e) ⟹ (wtpd_expr E e)"
by (simp only: wtpd_stmt_def wtpd_expr_def, erule wt_stmt.cases, auto)

lemma wtpd_stmt_comp: "wtpd_stmt E (s1;; s2) ⟹
   (wtpd_stmt E s1) &  (wtpd_stmt E s2)"
by (simp only: wtpd_stmt_def wtpd_expr_def, erule wt_stmt.cases, auto)
```

```
lemma wtpd_stmt_cond: "wtpd_stmt E (If(e) s1 Else s2) ⟹
   (wtpd_expr E e) & (wtpd_stmt E s1) &  (wtpd_stmt E s2)
  & (E⊢e::PrimT Boolean)"
by (simp only: wtpd_stmt_def wtpd_expr_def, erule wt_stmt.cases, auto)

lemma wtpd_stmt_loop: "wtpd_stmt E (While(e) s) ⟹
   (wtpd_expr E e) & (wtpd_stmt E s) & (E⊢e::PrimT Boolean)"
by (simp only: wtpd_stmt_def wtpd_expr_def, erule wt_stmt.cases, auto)

lemma wtpd_expr_call: "wtpd_expr E ({C}a..mn({pTs'}ps))
  ⟹ (wtpd_expr E a) & (wtpd_exprs E ps)
  & (length ps = length pTs') & (E⊢a::Class C)
  & (∃ pTs md rT.
       E⊢ps[::]pTs & max_spec (prg E) C (mn, pTs) = {((md,rT),pTs')})"
apply (simp only: wtpd_expr_def wtpd_exprs_def)
apply (erule exE)
apply (ind_cases "E ⊢ {C}a..mn( {pTs'}ps) :: T" for T)
apply (auto simp: max_spec_preserves_length)
done

lemma wtpd_blk:
  "⟦ method (G, D) (md, pTs) = Some (D, rT, (pns, lvars, blk, res));
  wf_prog wf_java_mdecl G; is_class G D ⟧
 ⟹ wtpd_stmt (env_of_jmb G D (md, pTs)) blk"
apply (simp add: wtpd_stmt_def env_of_jmb_def)
apply (frule_tac P="%D (md, pTs) (rT, (pns, lvars, blk, res)). (G, map_of lvars(pns[↦]pTs)(This↦Class
D)) ⊢ blk √ " in method_preserves)
apply (auto simp: wf_mdecl_def wf_java_mdecl_def)
done

lemma wtpd_res:
  "⟦ method (G, D) (md, pTs) = Some (D, rT, (pns, lvars, blk, res));
  wf_prog wf_java_mdecl G; is_class G D ⟧
 ⟹ wtpd_expr (env_of_jmb G D (md, pTs)) res"
apply (simp add: wtpd_expr_def env_of_jmb_def)
apply (frule_tac P="%D (md, pTs) (rT, (pns, lvars, blk, res)). ∃T. (G, map_of lvars(pns[↦]pTs)(This↦C
D)) ⊢ res :: T " in method_preserves)
apply (auto simp: wf_mdecl_def wf_java_mdecl_def)
done



lemma evals_preserves_length:
  "G⊢ xs -es[≻]vs-> (None, s) ⟹ length es = length vs"
apply (subgoal_tac
  "∀ xs'. (G ⊢ xk -xj≻xi-> xh ⟶ True) &
  (G⊢ xs -es[≻]vs-> xs' ⟶  (∃ s. (xs' = (None, s))) ⟶
  length es = length vs) &
  (G ⊢ xc -xb-> xa ⟶ True)")
apply blast
apply (rule allI)
```

```
apply (rule Eval.eval_evals_exec.induct)
apply auto
done

lemma progression_Eq : "{G, C, S} ⊢
  {hp, (v2 # v1 # os), lvars}
  >- [Ifcmpeq 3, LitPush (Bool False), Goto 2, LitPush (Bool True)] →
  {hp, (Bool (v1 = v2) # os), lvars}"
apply (case_tac "v1 = v2")

apply (rule_tac ?instrs1.0 = "[LitPush (Bool True)]" in jump_fwd_progression)
apply (auto simp: nat_add_distrib)
apply (rule progression_one_step) apply simp

apply (rule progression_one_step [THEN HOL.refl [THEN progression_transitive], simplified])
apply auto
apply (rule progression_one_step [THEN HOL.refl [THEN progression_transitive], simplified])

apply auto
apply (rule_tac ?instrs1.0 = "[]" in jump_fwd_progression)
apply (auto simp: nat_add_distrib intro: progression_refl)
done

declare split_paired_All [simp del] split_paired_Ex [simp del]
declaration {* K (Simplifier.map_ss (fn ss => ss delloop "split_all_tac")) *}

lemma distinct_method: "⟦ wf_java_prog G; is_class G C;
  method (G, C) S = Some (D, rT, pns, lvars, blk, res) ⟧ ⟹
  distinct (gjmb_plns (gmb G C S))"
apply (frule method_wf_mdecl [THEN conjunct2],  assumption, assumption)
apply (case_tac S)
apply (simp add: wf_mdecl_def wf_java_mdecl_def galldefs distinct_append)
apply (simp add: unique_def map_of_in_set)
apply blast
done

lemma distinct_method_if_class_sig_defined :
  "⟦ wf_java_prog G; class_sig_defined G C S ⟧ ⟹
  distinct (gjmb_plns (gmb G C S))"
by (auto intro: distinct_method simp: class_sig_defined_def)
```

lemma *method_yields_wf_java_mdecl: "⟦ wf_java_prog G; is_class G C;*
method (G, C) S = Some (D, rT, pns, lvars, blk, res) ⟧ ⟹
wf_java_mdecl G D (S,rT,(pns,lvars,blk,res))"
apply *(frule method_wf_mdecl)*
apply *(auto simp: wf_mdecl_def)*
done

lemma *progression_lvar_init_aux [rule_format (no_asm)]: "*
∀ zs prfx lvals lvars0.
lvars0 = (zs @ lvars) ⟶
(disjoint_varnames pns lvars0 ⟶
(length lvars = length lvals) ⟶
(Suc(length pns + length zs) = length prfx) ⟶
({cG, D, S} ⊢
{h, os, (prfx @ lvals)}
>- (concat (map (compInit (pns, lvars0, blk, res)) lvars)) →
{h, os, (prfx @ (map (λp. (default_val (snd p))) lvars))}))"
apply *simp*
apply *(induct lvars)*
apply *(clarsimp, rule progression_refl)*
apply *(intro strip)*
apply *(case_tac lvals)* **apply** *simp*
apply *(simp (no_asm_simp))*

apply *(rule_tac ?lvars1.0 = "(prfx @ [default_val (snd a)]) @ list"* **in** *progression_transitive, rule HOL.refl)*
apply *(case_tac a)* **apply** *(simp (no_asm_simp) add: compInit_def)*
apply *(rule_tac ?instrs0.0 = "[load_default_val b]"* **in** *progression_transitive, simp)*
apply *(rule progression_one_step)*
apply *(simp (no_asm_simp) add: load_default_val_def)*
apply *(rule conjI, simp)+* **apply** *(rule HOL.refl)*

apply *(rule progression_one_step)*
apply *(simp (no_asm_simp))*
apply *(rule conjI, simp)+*
apply *(simp add: index_of_var2)*
apply *(drule_tac x="zs @ [a]"* **in** *spec)*
apply *(drule mp, simp)*
apply *(drule_tac x="(prfx @ [default_val (snd a)])"* **in** *spec)*
apply *auto*
done

lemma *progression_lvar_init [rule_format (no_asm)]:*
"⟦ wf_java_prog G; is_class G C;
method (G, C) S = Some (D, rT, pns, lvars, blk, res) ⟧ ⟹
length pns = length pvs ⟶
(∀ lvals.
length lvars = length lvals ⟶
{cG, D, S} ⊢
{h, os, (a' # pvs @ lvals)}

```
   >- (compInitLvars (pns, lvars, blk, res) lvars) →
   {h, os, (locvars_xstate G C S (Norm (h, init_vars lvars(pns[↦]pvs)(This↦a'))))})"
apply (simp only: compInitLvars_def)
apply (frule method_yields_wf_java_mdecl, assumption, assumption)

apply (simp only: wf_java_mdecl_def)
apply (subgoal_tac "(∀y∈set pns. y ∉ set (map fst lvars))")
apply (simp add: init_vars_def locvars_xstate_def locvars_locals_def galldefs unique_def
split_def map_of_map_as_map_upd del: map_map)
apply (intro strip)
apply (simp (no_asm_simp) only: append_Cons [THEN sym])
apply (rule progression_lvar_init_aux)
apply (auto simp: unique_def map_of_in_set disjoint_varnames_def)
done
```

```
lemma state_ok_eval: "⟦xs::≼E; wf_java_prog (prg E); wtpd_expr E e;
  (prg E)⊢xs -e≻v -> xs'⟧ ⟹  xs'::≼E"
apply (simp only: wtpd_expr_def)
apply (erule exE)
apply (case_tac xs', case_tac xs)
apply (auto intro: eval_type_sound [THEN conjunct1])
done

lemma state_ok_evals: "⟦xs::≼E; wf_java_prog (prg E); wtpd_exprs E es;
  prg E ⊢ xs -es[≻]vs-> xs'⟧ ⟹ xs'::≼E"
apply (simp only: wtpd_exprs_def)
apply (erule exE)
apply (case_tac xs) apply (case_tac xs')
apply (auto intro: evals_type_sound [THEN conjunct1])
done

lemma state_ok_exec: "⟦xs::≼E; wf_java_prog (prg E); wtpd_stmt E st;
  prg E ⊢ xs -st-> xs'⟧ ⟹  xs'::≼E"
apply (simp only: wtpd_stmt_def)
apply (case_tac xs', case_tac xs)
apply (auto dest: exec_type_sound)
done


lemma state_ok_init:
  "⟦ wf_java_prog G; (x, h, l)::≼(env_of_jmb G C S);
  is_class G dynT;
  method (G, dynT) (mn, pTs) = Some (md, rT, pns, lvars, blk, res);
  list_all2 (conf G h) pvs pTs; G,h ⊢ a' ::≼ Class md⟧
⟹
(np a' x, h, init_vars lvars(pns[↦]pvs)(This↦a'))::≼(env_of_jmb G md (mn, pTs))"
apply (frule wf_prog_ws_prog)
apply (frule method_in_md [THEN conjunct2], assumption+)
apply (frule method_yields_wf_java_mdecl, assumption, assumption)
```

```
apply (simp add: env_of_jmb_def gs_def conforms_def split_beta)
apply (simp add: wf_java_mdecl_def)
apply (rule conjI)
apply (rule lconf_ext)
apply (rule lconf_ext_list)
apply (rule lconf_init_vars)
apply (auto dest: Ball_set_table)
apply (simp add: np_def xconf_raise_if)
done


lemma ty_exprs_list_all2 [rule_format (no_asm)]:
  "(∀ Ts. (E ⊢ es [::] Ts) = list_all2 (λe T. E ⊢ e :: T) es Ts)"
apply (rule list.induct)
apply simp
apply (rule allI)
apply (rule iffI)
  apply (ind_cases "E ⊢ [] [::] Ts" for Ts, assumption)
  apply simp apply (rule WellType.Nil)
apply (simp add: list_all2_Cons1)
apply (rule allI)
apply (rule iffI)
  apply (rename_tac a exs Ts)
  apply (ind_cases "E ⊢ a # exs  [::] Ts" for a exs Ts) apply blast
  apply (auto intro: WellType.Cons)
done


lemma conf_bool: "G,h ⊢ v::≼PrimT Boolean ⟹ ∃ b. v = Bool b"
apply (simp add: conf_def)
apply (erule exE)
apply (case_tac v)
apply (auto elim: widen.cases)
done


lemma class_expr_is_class: "⟦E ⊢ e :: Class C; ws_prog (prg E)⟧
  ⟹ is_class (prg E) C"
by (case_tac E, auto dest: ty_expr_is_type)


lemma max_spec_widen: "max_spec G C (mn, pTs) = {((md,rT),pTs')} ⟹
  list_all2 (λ T T'. G ⊢ T ≼ T') pTs pTs'"
by (blast dest: singleton_in_set max_spec2appl_meths appl_methsD)


lemma eval_conf: "⟦G ⊢ s -e≻v-> s'; wf_java_prog G; s::≼E;
  E⊢e::T; gx s' = None; prg E = G ⟧
  ⟹ G,gh s'⊢v::≼T"
apply (simp add: gh_def)
apply (rule_tac x3="fst s" and s3="snd s"and x'3="fst s'"
  in eval_type_sound [THEN conjunct2 [THEN conjunct1 [THEN mp]], simplified])
apply assumption+
apply (simp (no_asm_use) add: surjective_pairing [THEN sym])
```

```
apply (simp only: surjective_pairing [THEN sym])
apply (auto simp add: gs_def gx_def)
done

lemma evals_preserves_conf:
  "⟦ G⊢ s -es[≻]vs-> s'; G,gh s ⊢ t ::≼ T; E ⊢es[::]Ts;
  wf_java_prog G; s::≼E;
  prg E = G ⟧ ⟹ G,gh s' ⊢ t ::≼ T"
apply (subgoal_tac "gh s≤| gh s'")
apply (frule conf_hext, assumption, assumption)
apply (frule eval_evals_exec_type_sound [THEN conjunct2 [THEN conjunct1 [THEN mp]]])
apply (subgoal_tac "G ⊢ (gx s, (gh s, gl s)) -es[≻]vs-> (gx s', (gh s', gl s'))")
apply assumption
apply (simp add: gx_def gh_def gl_def surjective_pairing [THEN sym])
apply (case_tac E)
apply (simp add: gx_def gs_def gh_def gl_def surjective_pairing [THEN sym])
done

lemma eval_of_class: "⟦ G ⊢ s -e≻a'-> s'; E ⊢ e :: Class C;
  wf_java_prog G; s::≼E; gx s'=None; a' ≠ Null; G=prg E⟧
  ⟹ (∃ lc. a' = Addr lc)"
apply (case_tac s, case_tac s', simp)
apply (frule eval_type_sound, (simp add: gs_def)+)
apply (case_tac a')
apply (auto simp: conf_def)
done


lemma dynT_subcls:
  "⟦ a' ≠ Null; G,h⊢a'::≼ Class C; dynT = fst (the (h (the_Addr a')));
  is_class G dynT; ws_prog G ⟧ ⟹ G⊢dynT ≼C C"
apply (case_tac "C = Object")
apply (simp, rule subcls_C_Object, assumption+)
apply simp
apply (frule non_np_objD, auto)
done


lemma method_defined: "⟦
  m = the (method (G, dynT) (mn, pTs));
  dynT = fst (the (h a)); is_class G dynT; wf_java_prog G;
  a' ≠ Null; G,h⊢a'::≼ Class C; a = the_Addr a';
  ∃pTsa md rT. max_spec G C (mn, pTsa) = {((md, rT), pTs)} ⟧
⟹ (method (G, dynT) (mn, pTs)) = Some m"
apply (erule exE)+
apply (drule singleton_in_set, drule max_spec2appl_meths)
apply (simp add: appl_methds_def)
apply ((erule exE)+, (erule conjE)+, (erule exE)+)
apply (drule widen_methd)
apply assumption
apply (rule dynT_subcls) apply assumption+ apply simp apply simp
apply (erule exE)+ apply simp
done
```

```
theorem compiler_correctness:
  "wf_java_prog G ⟹
  (G ⊢ xs -ex≻val-> xs' ⟶
  gx xs = None ⟶ gx xs' = None ⟶
  (∀ os CL S.
  (class_sig_defined G CL S) ⟶
  (wtpd_expr (env_of_jmb G CL S) ex) ⟶
  (xs ::≼ (env_of_jmb G CL S)) ⟶
  ( {TranslComp.comp G, CL, S} ⊢
    {gh xs, os, (locvars_xstate G CL S xs)}
    >- (compExpr (gmb G CL S) ex) →
    {gh xs', val#os, locvars_xstate G CL S xs'}))) ∧

 (G ⊢ xs -exs[≻]vals-> xs' ⟶
  gx xs = None ⟶ gx xs' = None ⟶
  (∀ os CL S.
  (class_sig_defined G CL S) ⟶
  (wtpd_exprs (env_of_jmb G CL S) exs) ⟶
  (xs::≼(env_of_jmb G CL S)) ⟶
  ( {TranslComp.comp G, CL, S} ⊢
    {gh xs, os, (locvars_xstate G CL S xs)}
    >- (compExprs (gmb G CL S) exs) →
    {gh xs', (rev vals)@os, (locvars_xstate G CL S xs')}))) ∧

  (G ⊢ xs -st-> xs' ⟶
   gx xs = None ⟶ gx xs' = None ⟶
  (∀ os CL S.
  (class_sig_defined G CL S) ⟶
  (wtpd_stmt (env_of_jmb G CL S) st) ⟶
  (xs::≼(env_of_jmb G CL S)) ⟶
  ( {TranslComp.comp G, CL, S} ⊢
    {gh xs, os, (locvars_xstate G CL S xs)}
    >- (compStmt (gmb G CL S) st) →
    {gh xs', os, (locvars_xstate G CL S xs')})))"
apply (rule Eval.eval_evals_exec.induct)


apply simp


apply clarify
apply (frule wf_prog_ws_prog [THEN wf_subcls1])
apply (simp add: c_hupd_hp_invariant)
apply (rule progression_one_step)
apply (rotate_tac 1, drule sym)
apply (simp add: locvars_xstate_def locvars_locals_def comp_fields)
```

```
apply (intro allI impI)
apply simp
apply (frule raise_if_NoneD)
apply (frule wtpd_expr_cast)
apply simp
apply (rule_tac ?instrs0.0 = "(compExpr (gmb G CL S) e)" in progression_transitive, simp)
apply blast
apply (rule progression_one_step)
apply (simp add: raise_system_xcpt_def  gh_def comp_cast_ok)
```

```
apply simp
apply (intro strip)
apply (rule progression_one_step)
apply simp
```

```
apply (intro allI impI)
apply (frule_tac xs=s1 in eval_xcpt, assumption)
apply (frule wtpd_expr_binop)

apply (frule_tac e=e1 in state_ok_eval) apply (simp (no_asm_simp)) apply simp apply (simp
(no_asm_use) only: env_of_jmb_fst)
```

```
apply (simp (no_asm_use) only: compExpr_compExprs.simps)
apply (rule_tac ?instrs0.0 = "compExpr (gmb G CL S) e1" in progression_transitive, simp)
apply blast
apply (rule_tac ?instrs0.0 = "compExpr (gmb G CL S) e2" in progression_transitive, simp)
apply blast
apply (case_tac bop)
    apply simp apply (rule progression_Eq)
   apply simp apply (rule progression_one_step) apply simp
```

```
apply simp
apply (intro strip)
apply (rule progression_one_step)
apply (simp add: locvars_xstate_def locvars_locals_def)
apply (frule wtpd_expr_lacc)
apply assumption
apply (simp add: gl_def)
apply (erule select_at_index)
```

```
apply (intro allI impI)
apply (frule wtpd_expr_lass, erule conjE, erule conjE)
apply (simp add: compExpr_compExprs.simps)
```

apply *(rule_tac ?instrs0.0 = "(compExpr (gmb G CL S) e)"* **in** *progression_transitive, rule HOL.refl)*
apply *blast*

apply *(rule_tac ?instrs0.0 = "[Dup]"* **in** *progression_transitive, simp)*
apply *(rule progression_one_step)*
apply *(simp add: gh_def)*
apply *(rule conjI, simp)+* **apply** *simp*
apply *(rule progression_one_step)*
apply *(simp add: gh_def)*

apply *(frule wtpd_expr_lacc)* **apply** *assumption*
apply *(rule update_at_index)*
apply *(rule distinct_method_if_class_sig_defined)* **apply** *assumption*
apply *assumption* **apply** *simp* **apply** *assumption*

apply *(intro allI impI)*

apply *(simp (no_asm_use) only: gx_conv, frule np_NoneD)*
apply *(frule wtpd_expr_facc)*

apply *(simp (no_asm_use) only: compExpr_compExprs.simps)*
apply *(rule_tac ?instrs0.0 = "(compExpr (gmb G CL S) e)"* **in** *progression_transitive, rule HOL.refl)*
apply *blast*
apply *(rule progression_one_step)*
apply *(simp add: gh_def)*
apply *(case_tac "(the (fst s1 (the_Addr a')))")*
apply *(simp add: raise_system_xcpt_def)*

apply *(intro allI impI)*
apply *(frule wtpd_expr_fass)* **apply** *(erule conjE)* **apply** *(frule wtpd_expr_facc)*
apply *(simp only: c_hupd_xcpt_invariant)*

apply *(frule_tac xs="(np a' x1, s1)"* **in** *eval_xcpt)*
apply *(simp only: gx_conv, simp only: gx_conv, frule np_NoneD, erule conjE)*

apply *(frule_tac e=e1* **in** *state_ok_eval)* **apply** *(simp (no_asm_simp) only: env_of_jmb_fst)*

apply *assumption* **apply** *(simp (no_asm_use) only: env_of_jmb_fst)*

apply *(simp only: compExpr_compExprs.simps)*

apply *(rule_tac ?instrs0.0 = "(compExpr (gmb G CL S) e1)"* **in** *progression_transitive, rule HOL.refl)*
apply *fast*
apply *(rule_tac ?instrs0.0 = "(compExpr (gmb G CL S) e2)"* **in** *progression_transitive, rule*

```
HOL.refl)
apply fast
apply (rule_tac ?instrs0.0 = "[Dup_x1]" and ?instrs1.0 = "[Putfield fn T]" in progression_tran
simp)

   apply (rule progression_one_step)
   apply (simp add: gh_def)
   apply (rule conjI, simp)+ apply simp

   apply (rule progression_one_step)
   apply simp
   apply (case_tac "(the (fst s2 (the_Addr a')))")
   apply (simp add: c_hupd_hp_invariant)
   apply (case_tac s2)
   apply (simp add: c_hupd_conv raise_system_xcpt_def)
   apply (rule locvars_xstate_par_dep, rule HOL.refl)

defer

apply simp

apply (simp add: compExpr_compExprs.simps)
apply (intro strip)
apply (rule progression_refl)

apply (intro allI impI)
apply (frule_tac xs=s1 in evals_xcpt, simp only: gx_conv)
apply (frule wtpd_exprs_cons)

apply (frule_tac e=e in state_ok_eval) apply (simp (no_asm_simp) only: env_of_jmb_fst)
apply simp apply (simp (no_asm_use) only: env_of_jmb_fst)

apply simp
apply (rule_tac ?instrs0.0 = "(compExpr (gmb G CL S) e)" in progression_transitive, rule
HOL.refl)
apply fast
apply fast

apply simp

apply (intro allI impI)
apply simp
apply (rule progression_refl)
```

apply *(intro allI impI)*
apply *(frule wtpd_stmt_expr)*
apply *simp*
apply *(rule_tac ?instrs0.0 = "(compExpr (gmb G CL S) e)"* **in** *progression_transitive, rule HOL.refl)*
apply *fast*
apply *(rule progression_one_step)*
apply *simp*

apply *(intro allI impI)*
apply *(frule_tac xs=s1* **in** *exec_xcpt, assumption)*
apply *(frule wtpd_stmt_comp)*

apply *(frule_tac st=c1* **in** *state_ok_exec)* **apply** *(simp (no_asm_simp) only: env_of_jmb_fst)*
apply *simp* **apply** *(simp (no_asm_use) only: env_of_jmb_fst)*

apply *simp*
apply *(rule_tac ?instrs0.0 = "(compStmt (gmb G CL S) c1)"* **in** *progression_transitive, rule HOL.refl)*
apply *fast*
apply *fast*

apply *(intro allI impI)*
apply *(frule_tac xs=s1* **in** *exec_xcpt, assumption)*
apply *(frule wtpd_stmt_cond, (erule conjE)+)*

apply *(frule_tac e=e* **in** *state_ok_eval)*
apply *(simp (no_asm_simp) only: env_of_jmb_fst)*
apply *assumption*
apply *(simp (no_asm_use) only: env_of_jmb_fst)*

apply *(frule eval_conf, assumption+, rule env_of_jmb_fst)*
apply *(frule conf_bool)*
apply *(erule exE)*

apply *simp*
apply *(rule_tac ?instrs0.0 = "[LitPush (Bool False)]"* **in** *progression_transitive, simp (no_asm_simp))*
apply *(rule progression_one_step, simp)*
apply *(rule conjI, rule HOL.refl)+* **apply** *(rule HOL.refl)*

apply *(rule_tac ?instrs0.0 = "compExpr (gmb G CL S) e"* **in** *progression_transitive, rule HOL.refl)*
apply *fast*

apply *(case_tac b)*

apply *simp*
apply *(rule_tac ?instrs0.0 = "[Ifcmpeq (2 + int (length (compStmt (gmb G CL S) c1)))]"* **in** *progression_transitive, simp)*

apply *(rule progression_one_step)* **apply** *simp*
apply *(rule conjI, rule HOL.refl)+* **apply** *(rule HOL.refl)*
apply *(rule_tac ?instrs0.0 = "(compStmt (gmb G CL S) c1)"* **in** *progression_transitive, simp)*
apply *fast*
apply *(rule_tac ?instrs1.0 = "[]"* **in** *jump_fwd_progression)*
apply *(simp, rule conjI, (rule HOL.refl)+)*
apply *simp* **apply** *(rule conjI, simp)* **apply** *(simp add: nat_add_distrib)*
apply *(rule progression_refl)*

apply *simp*
apply *(rule_tac ?instrs1.0 = "compStmt (gmb G CL S) c2"* **in** *jump_fwd_progression)*
apply *(simp, rule conjI, (rule HOL.refl)+)*
apply *(simp, rule conjI, rule HOL.refl, simp add: nat_add_distrib)*
apply *fast*

apply *(intro allI impI)*
apply *(frule wtpd_stmt_loop, (erule conjE)+)*

apply *(frule eval_conf, assumption+, rule env_of_jmb_fst)*
apply *(frule conf_bool)*
apply *(erule exE)*
apply *(case_tac b)*

apply *simp*

apply *simp*

apply *(rule_tac ?instrs0.0 = "[LitPush (Bool False)]"* **in** *progression_transitive, simp (no_asm_simp))*
apply *(rule progression_one_step)*
  apply *simp*
  apply *(rule conjI, rule HOL.refl)+* **apply** *(rule HOL.refl)*

apply *(rule_tac ?instrs0.0 = "compExpr (gmb G CL S) e"* **in** *progression_transitive, rule HOL.refl)*
apply *fast*
apply *(rule_tac ?instrs1.0 = "[]"* **in** *jump_fwd_progression)*
apply *(simp, rule conjI, rule HOL.refl, rule HOL.refl)*
apply *(simp, rule conjI, rule HOL.refl, simp add: nat_add_distrib)*
apply *(rule progression_refl)*

apply *(intro allI impI)*
apply *(frule_tac xs=s2* **in** *exec_xcpt, assumption)*
apply *(frule_tac xs=s1* **in** *exec_xcpt, assumption)*
apply *(frule wtpd_stmt_loop, (erule conjE)+)*

apply *(frule_tac e=e* **in** *state_ok_eval)* **apply** *(simp (no_asm_simp) only: env_of_jmb_fst)*
apply *simp* **apply** *(simp (no_asm_use) only: env_of_jmb_fst)*

apply *(frule_tac xs=s1* **and** *st=c* **in** *state_ok_exec)*
apply *(simp (no_asm_simp) only: env_of_jmb_fst)* **apply** *assumption* **apply** *(simp (no_asm_use) only: env_of_jmb_fst)*

apply *(frule eval_conf, assumption+, rule env_of_jmb_fst)*
apply *(frule conf_bool)*
apply *(erule exE)*

apply *simp*
apply *(rule jump_bwd_progression)*
apply *simp*
apply *(rule conjI, (rule HOL.refl)+)*

apply *(rule_tac ?instrs0.0 = "[LitPush (Bool False)]"* **in** *progression_transitive, simp (no_asm_simp))*
apply *(rule progression_one_step)*
  apply *simp*
  apply *(rule conjI, simp)+* **apply** *simp*

apply *(rule_tac ?instrs0.0 = "compExpr (gmb G CL S) e"* **in** *progression_transitive, rule HOL.refl)*
apply *fast*

apply *(case_tac b)*

apply *simp*

apply *(rule_tac ?instrs0.0 = "[Ifcmpeq (2 + int (length (compStmt (gmb G CL S) c)))]"* **in** *progression_transitive, simp)*
apply *(rule progression_one_step)* **apply** *simp*
apply *(rule conjI, rule HOL.refl)+* **apply** *(rule HOL.refl)*
apply *fast*

apply *simp*

apply *(rule jump_bwd_one_step)*
apply *simp*
apply *blast*

apply *(intro allI impI)*

apply *(frule_tac xs=s3* **in** *eval_xcpt, simp only: gx_conv)*
apply *(frule exec_xcpt, assumption, simp (no_asm_use) only: gx_conv, frule np_NoneD)*
apply *(frule evals_xcpt, simp only: gx_conv)*

apply *(frule wtpd_expr_call, (erule conjE)+)*

```
apply (frule_tac xs="Norm s0" and e=e in state_ok_eval)
apply (simp (no_asm_simp) only: env_of_jmb_fst, assumption, simp (no_asm_use) only: env_of_jmb_

apply (frule_tac xs=s1 and xs'="(x, h, l)" in state_ok_evals)
apply (simp (no_asm_simp) only: env_of_jmb_fst, assumption, simp only: env_of_jmb_fst)

apply (frule (5) eval_of_class, rule env_of_jmb_fst [THEN sym])
apply (subgoal_tac "G,h ⊢ a' ::≼ Class C")
apply (subgoal_tac "is_class G dynT")

apply (drule method_defined, assumption+)
apply (simp only: env_of_jmb_fst)
apply ((erule exE)+, erule conjE, (rule exI)+, assumption)

apply (subgoal_tac "is_class G md")
apply (subgoal_tac "G⊢Class dynT ≼ Class md")
apply (subgoal_tac " method (G, md) (mn, pTs) = Some (md, rT, pns, lvars, blk, res)")
apply (subgoal_tac "list_all2 (conf G h) pvs pTs")

apply (subgoal_tac "G,h ⊢ a' ::≼ Class dynT")
apply (frule (2) conf_widen)
apply (frule state_ok_init, assumption+)

apply (subgoal_tac "class_sig_defined G md (mn, pTs)")
apply (frule wtpd_blk, assumption, assumption)
apply (frule wtpd_res, assumption, assumption)
apply (subgoal_tac "s3::≼(env_of_jmb G md (mn, pTs))")

apply (subgoal_tac "method (TranslComp.comp G, md) (mn, pTs) =
          Some (md, rT, snd (snd (compMethod G md ((mn, pTs), rT, pns, lvars, blk, res))))")
prefer 2 apply (simp add: wf_prog_ws_prog [THEN comp_method])
apply (subgoal_tac "method (TranslComp.comp G, dynT) (mn, pTs) =
          Some (md, rT, snd (snd (compMethod G md ((mn, pTs), rT, pns, lvars, blk, res))))")
prefer 2 apply (simp add: wf_prog_ws_prog [THEN comp_method])
 apply (simp only: fst_conv snd_conv)

apply (frule method_preserves_length, assumption, assumption)

apply (frule evals_preserves_length [THEN sym])

apply (simp (no_asm_use) only: compExpr_compExprs.simps)
```

apply *(rule_tac ?instrs0.0 = "(compExpr (gmb G CL S) e)"* **in** *progression_transitive, rule HOL.refl)*
apply *fast*

apply *(rule_tac ?instrs0.0 = "compExprs (gmb G CL S) ps"* **in** *progression_transitive, rule HOL.refl)*
apply *fast*

apply *(rule progression_call)*
apply *(intro allI impI conjI)*

apply *(simp (no_asm_use) only: exec_instr.simps)*
apply *(erule thin_rl, erule thin_rl, erule thin_rl)*
apply *(simp add: compMethod_def raise_system_xcpt_def)*
apply *(rule conjI, simp)+* **apply** *(rule HOL.refl)*

apply *(simp (no_asm_simp) add: gis_def gmb_def compMethod_def)*

apply *(rule_tac ?instrs0.0 = "(compInitLvars (pns, lvars, blk, res) lvars)"* **in** *progression_transitive, rule HOL.refl)*
apply *(rule_tac C=md* **in** *progression_lvar_init, assumption, assumption, assumption)*
apply *(simp (no_asm_simp))*
apply *(simp (no_asm_simp))*

apply *(rule_tac ?instrs0.0 = "compStmt (pns, lvars, blk, res) blk"* **in** *progression_transitive, rule HOL.refl)*
apply *(subgoal_tac "(pns, lvars, blk, res) = gmb G md (mn, pTs)")*
apply *(simp (no_asm_simp))*
apply *(simp only: gh_conv)*
apply *(drule mp [OF _ TrueI])+*
apply *(erule allE, erule allE, erule allE, erule impE, assumption)+*
apply *((erule impE, assumption)+, assumption)*

apply *(simp (no_asm_use))*
apply *(simp (no_asm_simp) add: gmb_def)*

apply *(subgoal_tac "(pns, lvars, blk, res) = gmb G md (mn, pTs)")*
apply *(simp (no_asm_simp))*
apply *(simp only: gh_conv)*
apply *((drule mp, rule TrueI)+, (drule spec)+, (drule mp, assumption)+, assumption)*
apply *(simp (no_asm_use))*
apply *(simp (no_asm_simp) add: gmb_def)*

apply *(simp (no_asm_use) add: gh_def locvars_xstate_def gl_def del: drop_append)*
apply *(subgoal_tac "rev pvs @ a' # os = (rev (a' # pvs)) @ os")*
apply *(simp only: drop_append)*

```
apply (simp (no_asm_simp))
apply (simp (no_asm_simp))

apply (rule_tac xs = "(np a' x, h, init_vars lvars(pns[↦]pvs)(This↦a'))" and st=blk
in state_ok_exec)
apply assumption apply (simp (no_asm_simp) only: env_of_jmb_fst)
apply assumption apply (simp (no_asm_use) only: env_of_jmb_fst)

apply (simp (no_asm_simp) add: class_sig_defined_def)

apply (frule non_npD) apply assumption
apply (erule exE)+ apply simp
apply (rule conf_obj_AddrI) apply simp
apply (rule conjI, (rule HOL.refl)+)
apply (rule widen_Class_Class [THEN iffD1], rule widen.refl)

apply (erule exE)+ apply (erule conjE)+
apply (rule_tac Ts="pTsa" in conf_list_gext_widen) apply assumption
apply (subgoal_tac "G ⊢ (gx s1, gs s1) -ps[≻]pvs-> (x, h, l)")
apply (frule_tac E="env_of_jmb G CL S" in evals_type_sound)
apply assumption+
apply (simp only: env_of_jmb_fst)
apply (simp add: conforms_def xconf_def gs_def)
apply simp
apply (simp (no_asm_use) only: gx_def gs_def surjective_pairing [THEN sym])
apply (simp (no_asm_use) only: ty_exprs_list_all2) apply simp
apply simp
apply (simp (no_asm_use) only: gx_def gs_def surjective_pairing [THEN sym])

    apply (rule max_spec_widen, simp only: env_of_jmb_fst)

apply (frule wf_prog_ws_prog [THEN method_in_md [THEN conjunct2]], assumption+)

apply (simp (no_asm_use) only: widen_Class_Class)
apply (rule method_wf_mdecl [THEN conjunct1], assumption+)

apply (rule wf_prog_ws_prog [THEN method_in_md [THEN conjunct1]], assumption+)

apply (frule non_npD) apply assumption
apply (erule exE)+ apply (erule conjE)+
apply simp
apply (rule subcls_is_class2) apply assumption
apply (frule class_expr_is_class) apply (simp only: env_of_jmb_fst)
apply (rule wf_prog_ws_prog, assumption)
```

```
apply (simp only: env_of_jmb_fst)


apply (simp only: wtpd_exprs_def, erule exE)
apply (frule evals_preserves_conf)
apply (rule eval_conf, assumption+)
apply (rule env_of_jmb_fst, assumption+)
apply (rule env_of_jmb_fst)
apply (simp only: gh_conv)
done


theorem compiler_correctness_eval: "
  ⟦ G ⊢ (None,hp,loc) -ex ≻ val-> (None,hp',loc');
  wf_java_prog G;
  class_sig_defined G C S;
  wtpd_expr (env_of_jmb G C S) ex;
  (None,hp,loc) ::≼ (env_of_jmb G C S) ⟧ ⟹
  {(TranslComp.comp G), C, S} ⊢
    {hp, os, (locvars_locals G C S loc)}
      >- (compExpr (gmb G C S) ex) →
    {hp', val#os, (locvars_locals G C S loc')}"
apply (frule compiler_correctness [THEN conjunct1])
apply (auto simp: gh_def gx_def gs_def gl_def locvars_xstate_def)
done

theorem compiler_correctness_exec: "
  ⟦ G ⊢ Norm (hp, loc) -st-> Norm (hp', loc');
  wf_java_prog G;
  class_sig_defined G C S;
  wtpd_stmt (env_of_jmb G C S) st;
  (None,hp,loc) ::≼ (env_of_jmb G C S) ⟧ ⟹
  {(TranslComp.comp G), C, S} ⊢
    {hp, os, (locvars_locals G C S loc)}
      >- (compStmt (gmb G C S) st) →
    {hp', os, (locvars_locals G C S loc')}"
apply (frule compiler_correctness [THEN conjunct2 [THEN conjunct2]])
apply (auto simp: gh_def gx_def gs_def gl_def locvars_xstate_def)
done




declare split_paired_All [simp] split_paired_Ex [simp]
declaration {* K (Simplifier.map_ss (fn ss => ss addloop ("split_all_tac", split_all_tac)))
*}

declare wf_prog_ws_prog [simp del]

end
```

```
theory TypeInf
imports "../J/WellType"
begin
```

```
lemma NewC_invers: "E⊢NewC C::T
  ⟹ T = Class C ∧ is_class (prg E) C"
by (erule ty_expr.cases, auto)

lemma Cast_invers: "E⊢Cast D e::T
  ⟹ ∃ C. T = Class D ∧ E⊢e::C ∧ is_class (prg E) D ∧ prg E⊢C≼? Class D"
by (erule ty_expr.cases, auto)

lemma Lit_invers: "E⊢Lit x::T
  ⟹ typeof (λv. None) x = Some T"
by (erule ty_expr.cases, auto)

lemma LAcc_invers: "E⊢LAcc v::T
  ⟹ localT E v = Some T ∧ is_type (prg E) T"
by (erule ty_expr.cases, auto)

lemma BinOp_invers: "E⊢BinOp bop e1 e2::T'
  ⟹ ∃ T. E⊢e1::T ∧ E⊢e2::T ∧
          (if bop = Eq then T' = PrimT Boolean
                      else T' = T ∧ T = PrimT Integer)"
by (erule ty_expr.cases, auto)

lemma LAss_invers: "E⊢v::=e::T'
  ⟹ ∃ T. v ~= This ∧ E⊢LAcc v::T ∧ E⊢e::T' ∧ prg E⊢T'≼T"
by (erule ty_expr.cases, auto)

lemma FAcc_invers: "E⊢{fd}a..fn::fT
  ⟹ ∃ C. E⊢a::Class C ∧ field (prg E,C) fn = Some (fd,fT)"
by (erule ty_expr.cases, auto)

lemma FAss_invers: "E⊢{fd}a..fn:=v::T'
⟹ ∃ T. E⊢{fd}a..fn::T ∧ E⊢v ::T' ∧ prg E⊢T'≼T"
by (erule ty_expr.cases, auto)

lemma Call_invers: "E⊢{C}a..mn({pTs'}ps)::rT
  ⟹ ∃ pTs md.
  E⊢a::Class C ∧ E⊢ps[::]pTs ∧ max_spec (prg E) C (mn, pTs) = {((md,rT),pTs')}"
by (erule ty_expr.cases, auto)


lemma Nil_invers: "E⊢[] [::] Ts ⟹ Ts = []"
by (erule ty_exprs.cases, auto)
```

```
lemma Cons_invers: "E⊢e#es[::]Ts ⟹
  ∃ T Ts'. Ts = T#Ts' ∧ E ⊢e::T ∧ E ⊢es[::]Ts'"
by (erule ty_exprs.cases, auto)


lemma Expr_invers: "E⊢Expr e√ ⟹ ∃ T. E⊢e::T"
by (erule wt_stmt.cases, auto)

lemma Comp_invers: "E⊢s1;; s2√ ⟹ E⊢s1√ ∧ E⊢s2√"
by (erule wt_stmt.cases, auto)

lemma Cond_invers: "E⊢If(e) s1 Else s2√
  ⟹ E⊢e::PrimT Boolean ∧ E⊢s1√ ∧ E⊢s2√"
by (erule wt_stmt.cases, auto)

lemma Loop_invers: "E⊢While(e) s√
  ⟹ E⊢e::PrimT Boolean ∧ E⊢s√"
by (erule wt_stmt.cases, auto)




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



lemma uniqueness_of_types: "
  (∀ (E::'a prog × (vname ⇒ ty option)) T1 T2.
  E⊢e :: T1 ⟶ E⊢e :: T2 ⟶ T1 = T2) ∧
  (∀ (E::'a prog × (vname ⇒ ty option)) Ts1 Ts2.
  E⊢es [::] Ts1 ⟶ E⊢es [::] Ts2 ⟶ Ts1 = Ts2)"
apply (rule expr.induct)


apply (intro strip)
apply (erule ty_expr.cases) apply simp+
apply (erule ty_expr.cases) apply simp+


apply (intro strip)
apply (erule ty_expr.cases) apply simp+
apply (erule ty_expr.cases) apply simp+


apply (intro strip)
apply (erule ty_expr.cases) apply simp+
apply (erule ty_expr.cases) apply simp+


apply (intro strip)
apply (case_tac binop)
```

```
apply (erule ty_expr.cases) apply simp+
apply (erule ty_expr.cases) apply simp+

apply (erule ty_expr.cases) apply simp+
apply (erule ty_expr.cases) apply simp+


apply (intro strip)
apply (erule ty_expr.cases) apply simp+
apply (erule ty_expr.cases) apply simp+


apply (intro strip)
apply (erule ty_expr.cases) apply simp+
apply (erule ty_expr.cases) apply simp+



apply (intro strip)
apply (drule FAcc_invers)+ apply (erule exE)+
  apply (subgoal_tac "C = Ca", simp) apply blast



apply (intro strip)
apply (drule FAss_invers)+ apply (erule exE)+ apply (erule conjE)+
apply (drule FAcc_invers)+ apply (erule exE)+ apply blast



apply (intro strip)
apply (drule Call_invers)+ apply (erule exE)+ apply (erule conjE)+
apply (subgoal_tac "pTs = pTsa", simp) apply blast


apply (intro strip)
apply (erule ty_exprs.cases)+ apply simp+

apply (intro strip)
apply (erule ty_exprs.cases, simp)
apply (erule ty_exprs.cases, simp)
apply (subgoal_tac "e = ea", simp) apply simp
done


lemma uniqueness_of_types_expr [rule_format (no_asm)]: "
  (∀ E T1 T2. E⊢e :: T1 ⟶ E⊢e :: T2 ⟶ T1 = T2)"
by (rule uniqueness_of_types [THEN conjunct1])

lemma uniqueness_of_types_exprs [rule_format (no_asm)]: "
  (∀ E Ts1 Ts2. E⊢es [::] Ts1 ⟶ E⊢es [::] Ts2 ⟶ Ts1 = Ts2)"
by (rule uniqueness_of_types [THEN conjunct2])
```

```
definition inferred_tp :: "[java_mb env, expr] ⇒ ty" where
  "inferred_tp E e == (SOME T. E⊢e :: T)"

definition inferred_tps :: "[java_mb env, expr list] ⇒ ty list" where
  "inferred_tps E es == (SOME Ts. E⊢es [::] Ts)"


lemma inferred_tp_wt: "E⊢e :: T ⟹ (inferred_tp E e) = T"
by (auto simp: inferred_tp_def intro: uniqueness_of_types_expr)

lemma inferred_tps_wt: "E⊢es [::] Ts ⟹ (inferred_tps E es) = Ts"
by (auto simp: inferred_tps_def intro: uniqueness_of_types_exprs)


end
```

Alternative definition of well-typing of bytecode, used in compiler type correctness proof

```
theory Altern
imports BVSpec
begin

definition check_type :: "jvm_prog ⇒ nat ⇒ nat ⇒ JVMType.state ⇒ bool" where
  "check_type G mxs mxr s ≡ s ∈ states G mxs mxr"

definition wt_instr_altern :: "[instr,jvm_prog,ty,method_type,nat,nat,p_count,
                exception_table,p_count] ⇒ bool" where
  "wt_instr_altern i G rT phi mxs mxr max_pc et pc ≡
  app i G mxs rT pc et (phi!pc) ∧
  check_type G mxs mxr (OK (phi!pc)) ∧
  (∀(pc',s') ∈ set (eff i G pc et (phi!pc)). pc' < max_pc ∧ G ⊢ s' <=' phi!pc')"

definition wt_method_altern :: "[jvm_prog,cname,ty list,ty,nat,nat,instr list,
                 exception_table,method_type] ⇒ bool" where
  "wt_method_altern G C pTs rT mxs mxl ins et phi ≡
  let max_pc = length ins in
  0 < max_pc ∧
  length phi = length ins ∧
  check_bounded ins et ∧
  wt_start G C pTs mxl phi ∧
  (∀pc. pc<max_pc ⟶ wt_instr_altern (ins!pc) G rT phi mxs (1+length pTs+mxl) max_pc
et pc)"


lemma wt_method_wt_method_altern :
  "wt_method G C pTs rT mxs mxl ins et phi ⟶ wt_method_altern G C pTs rT mxs mxl ins
et phi"
apply (simp add: wt_method_def wt_method_altern_def)
apply (intro strip)
apply clarify
apply (drule spec, drule mp, assumption)
```

```
apply (simp add: check_types_def wt_instr_def wt_instr_altern_def check_type_def)
apply (auto  intro: imageI)
done


lemma check_type_check_types [rule_format]:
  "(∀pc. pc < length phi ⟶ check_type G mxs mxr (OK (phi ! pc)))
  ⟶ check_types G mxs mxr (map OK phi)"
apply (induct phi)
apply (simp add: check_types_def)
apply (simp add: check_types_def)
apply clarify
apply (frule_tac x=0 in spec)
apply (simp add: check_type_def)
apply auto
done

lemma wt_method_altern_wt_method [rule_format]:
  "wt_method_altern G C pTs rT mxs mxl ins et phi ⟶ wt_method G C pTs rT mxs mxl ins
et phi"
apply (simp add: wt_method_def wt_method_altern_def)
apply (intro strip)
apply clarify
apply (rule conjI)

apply (rule check_type_check_types)
apply (simp add: wt_instr_altern_def)


apply (intro strip)
apply (drule spec, drule mp, assumption)
apply (simp add: wt_instr_def wt_instr_altern_def)
done


end



theory CorrCompTp
imports LemmasComp TypeInf "../BV/JVM" "../BV/Altern"
begin

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




definition inited_LT :: "[cname, ty list, (vname × ty) list] ⇒ locvars_type" where
  "inited_LT C pTs lvars == (OK (Class C))#((map OK pTs))@(map (Fun.comp OK snd) lvars)"

definition is_inited_LT :: "[cname, ty list, (vname × ty) list, locvars_type] ⇒ bool"
where
```

```
  "is_inited_LT C pTs lvars LT == (LT = (inited_LT C pTs lvars))"

definition local_env :: "[java_mb prog, cname, sig, vname list,(vname × ty) list] ⇒ java_mb
env" where
  "local_env G C S pns lvars ==
     let (mn, pTs) = S in (G,map_of lvars(pns[↦]pTs)(This↦Class C))"

lemma local_env_fst [simp]: "fst (local_env G C S pns lvars) = G"
by (simp add: local_env_def split_beta)


lemma wt_class_expr_is_class: "⟦ ws_prog G; E ⊢ expr :: Class cname;
  E = local_env G C (mn, pTs) pns lvars⟧
  ⟹ is_class G cname "
apply (subgoal_tac "((fst E), (snd E)) ⊢ expr :: Class cname")
apply (frule ty_expr_is_type) apply simp
apply simp apply (simp (no_asm_use))
done
```

4.24.6 index

```
lemma local_env_snd: "
  snd (local_env G C (mn, pTs) pns lvars) = map_of lvars(pns[↦]pTs)(This↦Class C)"
by (simp add: local_env_def)



lemma index_in_bounds: " length pns = length pTs ⟹
  snd (local_env G C (mn, pTs) pns lvars) vname = Some T
       ⟹ index (pns, lvars, blk, res) vname < length (inited_LT C pTs lvars)"
apply (simp add: local_env_snd index_def split_beta)
apply (case_tac "vname = This")
apply (simp add: inited_LT_def)
apply simp
apply (drule map_of_upds_SomeD)
apply (drule length_takeWhile)
apply (simp add: inited_LT_def)
done


lemma map_upds_append [rule_format (no_asm)]:
  "∀ x1s m. (length k1s = length x1s
  ⟶ m(k1s[↦]x1s)(k2s[↦]x2s) = m ((k1s@k2s)[↦](x1s@x2s)))"
apply (induct k1s)
apply simp
apply (intro strip)
apply (subgoal_tac "∃ x xr. x1s = x # xr")
apply clarify
apply simp

apply (case_tac x1s)
apply auto
done
```

```
lemma map_of_append [rule_format]:
  "∀ ys. (map_of ((rev xs) @ ys) = (map_of ys) ((map fst xs) [↦] (map snd xs)))"
apply (induct xs)
apply simp
apply (rule allI)
apply (drule_tac x="a # ys" in spec)
apply (simp only: rev.simps append_assoc append_Cons append_Nil
  map.simps map_of.simps map_upds_Cons hd.simps tl.simps)
done

lemma map_of_as_map_upds: "map_of (rev xs) = empty ((map fst xs) [↦] (map snd xs))"
by (rule map_of_append [of _ "[]", simplified])

lemma map_of_rev: "unique xs ⟹ map_of (rev xs) = map_of xs"
apply (induct xs)
apply simp
apply (simp add: unique_def map_of_append map_of_as_map_upds [THEN sym]
                 Map.map_of_append[symmetric] del:Map.map_of_append)
done

lemma map_upds_rev [rule_format]: "∀ xs. (distinct ks ⟶ length ks = length xs
  ⟶ m (rev ks [↦] rev xs) = m (ks [↦] xs))"
apply (induct ks)
apply simp
apply (intro strip)
apply (subgoal_tac "∃ x xr. xs = x # xr")
apply clarify
apply (drule_tac x=xr in spec)
apply (simp add: map_upds_append [THEN sym])

apply (case_tac xs)
apply auto
done

lemma map_upds_takeWhile [rule_format]:
  "∀ ks. (empty(rev ks[↦]rev xs)) k = Some x ⟶ length ks = length xs ⟶
    xs ! length (takeWhile (λz. z ≠ k) ks) = x"
apply (induct xs)
 apply simp
apply (intro strip)
apply (subgoal_tac "∃ k' kr. ks = k' # kr")
 apply (clarify)
 apply (drule_tac x=kr in spec)
 apply (simp only: rev.simps)
 apply (subgoal_tac "(empty(rev kr @ [k'][↦]rev xs @ [a])) = empty (rev kr[↦]rev xs)([k'][↦
  apply (simp split:split_if_asm)
 apply (simp add: map_upds_append [THEN sym])
apply (case_tac ks)
 apply auto
done


lemma local_env_inited_LT: "⟦ snd (local_env G C (mn, pTs) pns lvars) vname = Some T;
```

```
  length pns = length pTs; distinct pns; unique lvars ⟧
  ⟹ (inited_LT C pTs lvars ! index (pns, lvars, blk, res) vname) = OK T"
apply (simp add: local_env_snd index_def)
apply (case_tac "vname = This")
apply (simp add: inited_LT_def)
apply (simp add: inited_LT_def)
apply (simp (no_asm_simp) only: map_map [symmetric] map_append [symmetric] map.simps [symmetric])
apply (subgoal_tac "length (takeWhile (λz. z ≠ vname) (pns @ map fst lvars)) < length
(pTs @ map snd lvars)")
apply (simp (no_asm_simp) only: List.nth_map ok_val.simps)
apply (subgoal_tac "map_of lvars = map_of (map (λ p. (fst p, snd p)) lvars)")
apply (simp only:)
apply (subgoal_tac "distinct (map fst lvars)")
apply (frule_tac g=snd in AuxLemmas.map_of_map_as_map_upd)
apply (simp only:)
apply (simp add: map_upds_append)
apply (frule map_upds_SomeD)
apply (rule map_upds_takeWhile)
apply (simp (no_asm_simp))
apply (simp add: map_upds_append [THEN sym])
apply (simp add: map_upds_rev)


apply simp


apply (simp only: unique_def Fun.comp_def)


apply simp


apply (drule map_of_upds_SomeD)
apply (drule length_takeWhile)
apply simp
done


lemma inited_LT_at_index_no_err: " i < length (inited_LT C pTs lvars)
  ⟹ inited_LT C pTs lvars ! i ≠ Err"
apply (simp only: inited_LT_def)
apply (simp only: map_map [symmetric] map_append [symmetric] map.simps [symmetric] length_map)
apply (simp only: nth_map)
apply simp
done


lemma sup_loc_update_index: "
  ⟦ G ⊢ T ≼ T'; is_type G T'; length pns = length pTs; distinct pns; unique lvars;
  snd (local_env G C (mn, pTs) pns lvars) vname = Some T' ⟧
  ⟹
  comp G ⊢
  inited_LT C pTs lvars [index (pns, lvars, blk, res) vname := OK T] <=l
  inited_LT C pTs lvars"
```

```
apply (subgoal_tac " index (pns, lvars, blk, res) vname < length (inited_LT C pTs lvars)")
apply (frule_tac blk=blk and res=res in local_env_inited_LT, assumption+)
apply (rule sup_loc_trans)
apply (rule_tac b="OK T'" in sup_loc_update)
apply (simp add: comp_widen)
apply assumption
apply (rule sup_loc_refl)
apply (simp add: list_update_same_conv [THEN iffD2])

apply (rule index_in_bounds)
apply simp+
done
```

4.24.7 Preservation of ST and LT by compTpExpr / compTpStmt

```
lemma sttp_of_comb_nil [simp]: "sttp_of (comb_nil sttp) = sttp"
by (simp add: comb_nil_def)

lemma mt_of_comb_nil [simp]: "mt_of (comb_nil sttp) = []"
by (simp add: comb_nil_def)


lemma sttp_of_comb [simp]: "sttp_of ((f1 □ f2) sttp) = sttp_of (f2 (sttp_of (f1 sttp)))"
apply (case_tac "f1 sttp")
apply (case_tac "(f2 (sttp_of (f1 sttp)))")
apply (simp add: comb_def)
done

lemma mt_of_comb: "(mt_of ((f1 □ f2) sttp)) =
  (mt_of (f1 sttp)) @ (mt_of (f2 (sttp_of (f1 sttp))))"
by (simp add: comb_def split_beta)


lemma mt_of_comb_length [simp]: "⟦ n1 = length (mt_of (f1 sttp)); n1 ≤ n ⟧
  ⟹ (mt_of ((f1 □ f2) sttp) ! n) = (mt_of (f2 (sttp_of (f1 sttp))) ! (n - n1))"
by (simp add: comb_def nth_append split_beta)


lemma compTpExpr_Exprs_LT_ST: "
  ⟦jmb = (pns, lvars, blk, res);
  wf_prog wf_java_mdecl G;
  wf_java_mdecl G C ((mn, pTs), rT, jmb);
  E = local_env G C (mn, pTs) pns lvars ⟧
 ⟹
  (∀ ST LT T.
  E ⊢ ex :: T ⟶
  is_inited_LT C pTs lvars LT ⟶
  sttp_of (compTpExpr jmb G ex (ST, LT)) = (T # ST, LT))
 ∧
 (∀ ST LT Ts.
  E ⊢ exs [::] Ts ⟶
  is_inited_LT C pTs lvars LT ⟶
  sttp_of (compTpExprs jmb G exs (ST, LT)) = ((rev Ts) @ ST, LT))"
```

```
apply (rule expr.induct)




apply (intro strip)
apply (drule NewC_invers)
apply (simp add: pushST_def)


apply (intro strip)
apply (drule Cast_invers, clarify)
apply ((drule_tac x=ST in spec), (drule spec)+, (drule mp, assumption)+)
apply (simp add: replST_def split_beta)


apply (intro strip)
apply (drule Lit_invers)
apply (simp add: pushST_def)


apply (intro strip)
apply (drule BinOp_invers, clarify)
apply (drule_tac x=ST in spec)
apply (drule_tac x="Ta # ST" in spec)
apply ((drule spec)+, (drule mp, assumption)+)
  apply (case_tac binop)
  apply (simp (no_asm_simp))
    apply (simp (no_asm_simp) add: popST_def pushST_def)
  apply (simp)
    apply (simp (no_asm_simp) add: replST_def)



apply (intro strip)
apply (drule LAcc_invers)
apply (simp add: pushST_def is_inited_LT_def)
apply (simp add: wf_prog_def)
apply (frule wf_java_mdecl_disjoint_varnames)
  apply (simp add: disjoint_varnames_def)
apply (frule wf_java_mdecl_length_pTs_pns)
apply (erule conjE)+
apply (simp (no_asm_simp) add: local_env_inited_LT)


apply (intro strip)
apply (drule LAss_invers, clarify)
apply (drule LAcc_invers)
apply ((drule_tac x=ST in spec), (drule spec)+, (drule mp, assumption)+)
apply (simp add: popST_def dupST_def)


apply (intro strip)
apply (drule FAcc_invers, clarify)
```

apply *((drule_tac x=ST* **in** *spec), (drule spec)+, (drule mp, assumption)+)*
apply *(simp add: replST_def)*

apply *(subgoal_tac "is_class G Ca")*
apply *(subgoal_tac "is_class G cname ∧ field (G, cname) vname = Some (cname, T)")*
apply *simp*

apply *(rule field_in_fd)* **apply** *assumption+*

apply *(fast intro: wt_class_expr_is_class)*

apply *(intro strip)*
apply *(drule FAss_invers, clarify)*
apply *(drule FAcc_invers, clarify)*
apply *(drule_tac x=ST* **in** *spec)*
apply *(drule_tac x="Class Ca # ST"* **in** *spec)*
apply *((drule spec)+, (drule mp, assumption)+)*
apply *(simp add: popST_def dup_x1ST_def)*

apply *(intro strip)*
apply *(drule Call_invers, clarify)*
apply *(drule_tac x=ST* **in** *spec)*
apply *(drule_tac x="Class cname # ST"* **in** *spec)*
apply *((drule spec)+, (drule mp, assumption)+)*
apply *(simp add: replST_def)*

apply *(intro strip)*
apply *(drule Nil_invers)*
apply *(simp add: comb_nil_def)*

apply *(intro strip)*
apply *(drule Cons_invers, clarify)*
apply *(drule_tac x=ST* **in** *spec)*
apply *(drule_tac x="T # ST"* **in** *spec)*
apply *((drule spec)+, (drule mp, assumption)+)*
apply *simp*

done

lemmas *compTpExpr_LT_ST [rule_format (no_asm)] =*
compTpExpr_Exprs_LT_ST [THEN conjunct1]

```
lemmas compTpExprs_LT_ST [rule_format (no_asm)] =
  compTpExpr_Exprs_LT_ST [THEN conjunct2]

lemma compTpStmt_LT_ST [rule_format (no_asm)]: "
  ⟦ jmb = (pns,lvars,blk,res);
  wf_prog wf_java_mdecl G;
  wf_java_mdecl G C ((mn, pTs), rT, jmb);
  E = (local_env G C (mn, pTs) pns lvars)⟧
⟹ (∀ ST LT.
  E ⊢ s√ ⟶
  (is_inited_LT C pTs lvars LT)
⟶ sttp_of (compTpStmt jmb G s (ST, LT)) = (ST, LT))"

apply (rule stmt.induct)


apply (intro strip)
apply simp


apply (intro strip)
apply (drule Expr_invers, erule exE)
apply (simp (no_asm_simp) add: compTpExpr_LT_ST)
apply (frule_tac ST=ST in compTpExpr_LT_ST, assumption+)
apply (simp add: popST_def)


apply (intro strip)
apply (drule Comp_invers, clarify)
apply (simp (no_asm_use))
apply simp


apply (intro strip)
apply (drule Cond_invers)
apply (erule conjE)+
apply (drule_tac x=ST in spec)
apply (drule_tac x=ST in spec)
apply (drule spec)+ apply (drule mp, assumption)+
apply (drule_tac ST="PrimT Boolean # ST" in compTpExpr_LT_ST, assumption+)
apply (simp add: popST_def pushST_def nochangeST_def)


apply (intro strip)
apply (drule Loop_invers)
apply (erule conjE)+
apply (drule_tac x=ST in spec)
apply (drule spec)+ apply (drule mp, assumption)+
apply (drule_tac ST="PrimT Boolean # ST" in compTpExpr_LT_ST, assumption+)
apply (simp add: popST_def pushST_def nochangeST_def)
done
```

lemma *compTpInit_LT_ST: "*
```
  sttp_of (compTpInit jmb (vn,ty) (ST, LT)) = (ST, LT[(index jmb vn):= OK ty])"
```
by *(simp add: compTpInit_def storeST_def pushST_def)*

lemma *compTpInitLvars_LT_ST_aux [rule_format (no_asm)]:*
```
  "∀ pre lvars_pre lvars0.
  jmb = (pns,lvars0,blk,res) ∧
  lvars0 = (lvars_pre @ lvars) ∧
  (length pns) + (length lvars_pre) + 1 = length pre ∧
  disjoint_varnames pns (lvars_pre @ lvars)
  ⟶
sttp_of (compTpInitLvars jmb lvars (ST, pre @ replicate (length lvars) Err))
    = (ST, pre @ map (Fun.comp OK snd) lvars)"
```
apply *(induct lvars)*
apply *simp*

apply *(intro strip)*
apply *(subgoal_tac "∃ vn ty. a = (vn, ty)")*
  prefer *2* **apply** *(simp (no_asm_simp))*
  apply *((erule exE)+, simp (no_asm_simp))*

apply *(drule_tac x="pre @ [OK ty]"* **in** *spec)*
apply *(drule_tac x="lvars_pre @ [a]"* **in** *spec)*
apply *(drule_tac x="lvars0"* **in** *spec)*
apply *(simp add: compTpInit_LT_ST index_of_var2)*
done

lemma *compTpInitLvars_LT_ST:*
```
  "⟦ jmb = (pns, lvars, blk, res); wf_java_mdecl G C ((mn, pTs), rT, jmb) ⟧
⟹(sttp_of (compTpInitLvars jmb lvars (ST, start_LT C pTs (length lvars)))
  = (ST, inited_LT C pTs lvars)"
```
apply *(simp add: start_LT_def inited_LT_def)*
apply *(simp only: append_Cons [THEN sym])*
apply *(rule compTpInitLvars_LT_ST_aux)*
apply *(auto dest: wf_java_mdecl_length_pTs_pns wf_java_mdecl_disjoint_varnames)*
done

lemma *max_of_list_elem: "x ∈ set xs ⟹ x ≤ (max_of_list xs)"*
by *(induct xs, auto intro: le_maxI1 simp: le_max_iff_disj max_of_list_def)*

lemma *max_of_list_sublist: "set xs ⊆ set ys*
  ⟹ (max_of_list xs) ≤ (max_of_list ys)"
by *(induct xs, auto dest: max_of_list_elem simp: max_of_list_def)*

lemma *max_of_list_append [simp]:*
  "max_of_list (xs @ ys) = max (max_of_list xs) (max_of_list ys)"
apply *(simp add: max_of_list_def)*
apply *(induct xs)*

```
apply simp
using [[linarith_split_limit = 0]]
apply simp
apply arith
done


lemma app_mono_mxs: "⟦ app i G mxs rT pc et s; mxs ≤ mxs' ⟧
  ⟹ app i G mxs' rT pc et s"
apply (case_tac s)
apply (simp add: app_def)
apply (case_tac i)
apply (auto intro: less_trans)
done



lemma err_mono [simp]: "A ⊆ B ⟹ err A ⊆ err B"
by (auto simp: err_def)

lemma opt_mono [simp]: "A ⊆ B ⟹ opt A ⊆ opt B"
by (auto simp: opt_def)


lemma states_mono: "⟦ mxs ≤ mxs' ⟧
  ⟹ states G mxs mxr ⊆ states G mxs' mxr"
apply (simp add: states_def JVMType.sl_def)
apply (simp add: Product.esl_def stk_esl_def reg_sl_def
  upto_esl_def Listn.sl_def Err.sl_def  JType.esl_def)
apply (simp add: Err.esl_def Err.le_def Listn.le_def)
apply (simp add: Product.le_def Product.sup_def Err.sup_def)
apply (simp add: Opt.esl_def Listn.sup_def)
apply (rule err_mono)
apply (rule opt_mono)
apply (rule Sigma_mono)
apply (rule Union_mono)
apply auto
done


lemma check_type_mono: "⟦ check_type G mxs mxr s; mxs ≤ mxs' ⟧
  ⟹ check_type G mxs' mxr s"
apply (simp add: check_type_def)
apply (frule_tac G=G and mxr=mxr in states_mono)
apply auto
done



lemma wt_instr_prefix: "
 ⟦ wt_instr_altern (bc ! pc) cG rT mt mxs mxr max_pc et pc;
  bc' = bc @ bc_post; mt' = mt @ mt_post;
  mxs ≤ mxs'; max_pc ≤ max_pc';
  pc < length bc; pc < length mt;
```

```
  max_pc = (length mt)⟧
⟹ wt_instr_altern (bc' ! pc) cG rT mt' mxs' mxr max_pc' et pc"
apply (simp add: wt_instr_altern_def nth_append)
apply (auto intro: app_mono_mxs check_type_mono)
done
```

```
lemma pc_succs_shift: "pc'∈set (succs i (pc'' + n))
 ⟹ ((pc' - n) ∈set (succs i pc''))"
apply (induct i)
apply simp_all
apply arith
done
```

```
lemma pc_succs_le: "⟦ pc' ∈ set (succs i (pc'' + n));
  ∀ b. ((i = (Goto b) ∨ i=(Ifcmpeq b)) ⟶ 0 ≤ (int pc'' + b)) ⟧
  ⟹ n ≤ pc'"
apply (induct i)
apply simp_all
apply arith
done
```

```
definition offset_xcentry :: "[nat, exception_entry] ⇒ exception_entry" where
  "offset_xcentry ==
      λ n (start_pc, end_pc, handler_pc, catch_type).
          (start_pc + n, end_pc + n, handler_pc + n, catch_type)"

definition offset_xctable :: "[nat, exception_table] ⇒ exception_table" where
  "offset_xctable n == (map (offset_xcentry n))"

lemma match_xcentry_offset [simp]: "
  match_exception_entry G cn (pc + n) (offset_xcentry n ee) =
  match_exception_entry G cn pc ee"
by (simp add: match_exception_entry_def offset_xcentry_def split_beta)

lemma match_xctable_offset: "
  (match_exception_table G cn (pc + n) (offset_xctable n et)) =
  (Option.map (λ pc'. pc' + n) (match_exception_table G cn pc et))"
apply (induct et)
apply (simp add: offset_xctable_def)+
apply (case_tac "match_exception_entry G cn pc a")
apply (simp add: offset_xcentry_def split_beta)+
done
```

```
lemma match_offset [simp]: "
```

```
  match G cn (pc + n) (offset_xctable n et) = match G cn pc et"
apply (induct et)
apply (simp add: offset_xctable_def)+
done

lemma match_any_offset [simp]: "
  match_any G (pc + n) (offset_xctable n et) = match_any G pc et"
apply (induct et)
apply (simp add: offset_xctable_def offset_xcentry_def split_beta)+
done

lemma app_mono_pc: "⟦ app i G mxs rT pc et s; pc'= pc + n ⟧
  ⟹ app i G mxs rT pc' (offset_xctable n et) s"
apply (case_tac s)
apply (simp add: app_def)
apply (case_tac i)
apply (auto)
done

abbreviation (input)
  empty_et :: exception_table
  where "empty_et == []"

lemma xcpt_names_Nil [simp]: "(xcpt_names (i, G, pc, [])) = []"
by (induct i, simp_all)

lemma xcpt_eff_Nil [simp]: "(xcpt_eff i G pc s []) = []"
by (simp add: xcpt_eff_def)

lemma app_jumps_lem: "⟦ app i cG mxs rT pc empty_et s; s=(Some st) ⟧
  ⟹  ∀ b. ((i = (Goto b) ∨ i=(Ifcmpeq b)) ⟶ 0 ≤ (int pc + b))"
apply (simp only:)
apply (induct i)
apply auto
done

lemma wt_instr_offset: "
 ⟦ ∀ pc'' < length mt.
    wt_instr_altern ((bc@bc_post) ! pc'') cG rT (mt@mt_post) mxs mxr max_pc empty_et pc'';

  bc' = bc_pre @ bc @ bc_post; mt' = mt_pre @ mt @ mt_post;
  length bc_pre = length mt_pre; length bc = length mt;
  length mt_pre ≤ pc; pc < length (mt_pre @ mt);
  mxs ≤ mxs'; max_pc + length mt_pre ≤ max_pc' ⟧
```

```
⟹ wt_instr_altern (bc' ! pc) cG rT mt' mxs' mxr max_pc' empty_et pc"

apply (simp add: wt_instr_altern_def)
apply (subgoal_tac "∃ pc''. pc = pc'' + length mt_pre", erule exE)
prefer 2 apply (rule_tac x="pc - length mt_pre" in exI, arith)

apply (drule_tac x=pc'' in spec)
apply (drule mp) apply arith
apply clarify

apply (rule conjI)

  apply (simp add: nth_append)
  apply (rule app_mono_mxs)
  apply (frule app_mono_pc) apply (rule HOL.refl) apply (simp add: offset_xctable_def)
  apply assumption+


apply (rule conjI)
apply (simp add: nth_append)
apply (rule  check_type_mono) apply assumption+


apply (intro ballI)
apply (subgoal_tac "∃ pc' s'. x = (pc', s')", (erule exE)+, simp)

apply (case_tac s')

apply (simp add: eff_def nth_append norm_eff_def)
apply (frule_tac x="(pc', None)" and  f=fst and b=pc' in rev_image_eqI)
  apply (simp (no_asm_simp))
  apply (simp only: fst_conv image_compose [THEN sym] Fun.comp_def)
  apply simp
  apply (frule pc_succs_shift)
apply (drule bspec, assumption)
apply arith


apply (drule_tac x="(pc' - length mt_pre, s')" in bspec)


apply (simp add: eff_def)

  apply (clarsimp simp: nth_append pc_succs_shift)


apply simp
  apply (subgoal_tac "length mt_pre ≤ pc'")
  apply (simp add: nth_append) apply arith


apply (simp add: eff_def xcpt_eff_def)
apply (clarsimp)
apply (rule pc_succs_le) apply assumption+
```

```
apply (subgoal_tac "∃ st. mt ! pc'' = Some st", erule exE)
 apply (rule_tac s="Some st" and st=st and cG=cG and mxs=mxs and rT=rT
        in app_jumps_lem)
  apply (simp add: nth_append)+

  apply (simp add: norm_eff_def Option.map_def nth_append)
  apply (case_tac "mt ! pc''")
apply simp+
done
```

```
definition start_sttp_resp_cons :: "[state_type ⇒ method_type × state_type] ⇒ bool"
where
  "start_sttp_resp_cons f ==
     (∀ sttp. let (mt', sttp') = (f sttp) in (∃mt'_rest. mt' = Some sttp # mt'_rest))"

definition start_sttp_resp :: "[state_type ⇒ method_type × state_type] ⇒ bool" where
  "start_sttp_resp f == (f = comb_nil) ∨ (start_sttp_resp_cons f)"

lemma start_sttp_resp_comb_nil [simp]: "start_sttp_resp comb_nil"
by (simp add: start_sttp_resp_def)

lemma start_sttp_resp_cons_comb_cons [simp]: "start_sttp_resp_cons f
  ⟹ start_sttp_resp_cons (f □ f')"
apply (simp add: start_sttp_resp_cons_def comb_def split_beta)
apply (rule allI)
apply (drule_tac x=sttp in spec)
apply auto
done

lemma start_sttp_resp_cons_comb_cons_r: "⟦ start_sttp_resp f; start_sttp_resp_cons f'⟧
  ⟹ start_sttp_resp_cons (f □ f')"
apply (simp add: start_sttp_resp_def)
apply (erule disjE)
apply simp+
done

lemma start_sttp_resp_cons_comb [simp]: "start_sttp_resp_cons f
  ⟹ start_sttp_resp (f □ f')"
by (simp add: start_sttp_resp_def)

lemma start_sttp_resp_comb: "⟦ start_sttp_resp f; start_sttp_resp f' ⟧
  ⟹ start_sttp_resp (f □ f')"
apply (simp add: start_sttp_resp_def)
apply (erule disjE)
apply simp
apply (erule disjE)
apply simp+
done

lemma start_sttp_resp_cons_nochangeST [simp]: "start_sttp_resp_cons nochangeST"
```

by *(simp add: start_sttp_resp_cons_def nochangeST_def)*

lemma *start_sttp_resp_cons_pushST [simp]: "start_sttp_resp_cons (pushST Ts)"*
by *(simp add: start_sttp_resp_cons_def pushST_def split_beta)*

lemma *start_sttp_resp_cons_dupST [simp]: "start_sttp_resp_cons dupST"*
by *(simp add: start_sttp_resp_cons_def dupST_def split_beta)*

lemma *start_sttp_resp_cons_dup_x1ST [simp]: "start_sttp_resp_cons dup_x1ST"*
by *(simp add: start_sttp_resp_cons_def dup_x1ST_def split_beta)*

lemma *start_sttp_resp_cons_popST [simp]: "start_sttp_resp_cons (popST n)"*
by *(simp add: start_sttp_resp_cons_def popST_def split_beta)*

lemma *start_sttp_resp_cons_replST [simp]: "start_sttp_resp_cons (replST n tp)"*
by *(simp add: start_sttp_resp_cons_def replST_def split_beta)*

lemma *start_sttp_resp_cons_storeST [simp]: "start_sttp_resp_cons (storeST i tp)"*
by *(simp add: start_sttp_resp_cons_def storeST_def split_beta)*

lemma *start_sttp_resp_cons_compTpExpr [simp]: "start_sttp_resp_cons (compTpExpr jmb G ex)"*
apply *(induct ex)*
apply *simp+*
apply *(simp add: start_sttp_resp_cons_def comb_def pushST_def split_beta)*
apply *simp+*
done

lemma *start_sttp_resp_cons_compTpInit [simp]: "start_sttp_resp_cons (compTpInit jmb lv)"*
by *(simp add: compTpInit_def split_beta)*

lemma *start_sttp_resp_nochangeST [simp]: "start_sttp_resp nochangeST"*
by *(simp add: start_sttp_resp_def)*

lemma *start_sttp_resp_pushST [simp]: "start_sttp_resp (pushST Ts)"*
by *(simp add: start_sttp_resp_def)*

lemma *start_sttp_resp_dupST [simp]: "start_sttp_resp dupST"*
by *(simp add: start_sttp_resp_def)*

lemma *start_sttp_resp_dup_x1ST [simp]: "start_sttp_resp dup_x1ST"*
by *(simp add: start_sttp_resp_def)*

lemma *start_sttp_resp_popST [simp]: "start_sttp_resp (popST n)"*
by *(simp add: start_sttp_resp_def)*

lemma *start_sttp_resp_replST [simp]: "start_sttp_resp (replST n tp)"*
by *(simp add: start_sttp_resp_def)*

lemma *start_sttp_resp_storeST [simp]: "start_sttp_resp (storeST i tp)"*
by *(simp add: start_sttp_resp_def)*

lemma *start_sttp_resp_compTpExpr [simp]: "start_sttp_resp (compTpExpr jmb G ex)"*

```
by (simp add: start_sttp_resp_def)

lemma start_sttp_resp_compTpExprs [simp]: "start_sttp_resp (compTpExprs jmb G exs)"
by (induct exs, (simp add: start_sttp_resp_comb)+)

lemma start_sttp_resp_compTpStmt [simp]: "start_sttp_resp (compTpStmt jmb G s)"
by (induct s, (simp add: start_sttp_resp_comb)+)

lemma start_sttp_resp_compTpInitLvars [simp]: "start_sttp_resp (compTpInitLvars jmb lvars)"
by (induct lvars, simp+)
```

4.24.8 length of compExpr/ compTpExprs

```
lemma length_comb [simp]:  "length (mt_of ((f1 □ f2) sttp)) =
  length (mt_of (f1 sttp)) + length (mt_of (f2 (sttp_of (f1 sttp))))"
by (simp add: comb_def split_beta)


lemma length_comb_nil [simp]: "length (mt_of (comb_nil sttp)) = 0"
by (simp add: comb_nil_def)

lemma length_nochangeST [simp]: "length (mt_of (nochangeST sttp)) = 1"
by (simp add: nochangeST_def)

lemma length_pushST [simp]: "length (mt_of (pushST Ts sttp)) = 1"
by (simp add: pushST_def split_beta)

lemma length_dupST [simp]: "length (mt_of (dupST sttp)) = 1"
by (simp add: dupST_def split_beta)

lemma length_dup_x1ST [simp]: "length (mt_of (dup_x1ST sttp)) = 1"
by (simp add: dup_x1ST_def split_beta)

lemma length_popST [simp]: "length (mt_of (popST n sttp)) = 1"
by (simp add: popST_def split_beta)

lemma length_replST [simp]: "length (mt_of (replST n tp sttp)) = 1"
by (simp add: replST_def split_beta)

lemma length_storeST [simp]: "length (mt_of (storeST i tp sttp)) = 1"
by (simp add: storeST_def split_beta)


lemma length_compTpExpr_Exprs [rule_format]: "
  (∀ sttp. (length (mt_of (compTpExpr jmb G ex sttp)) = length (compExpr jmb ex)))
  ∧ (∀ sttp. (length (mt_of (compTpExprs jmb G exs sttp)) = length (compExprs jmb exs)))"
apply (rule  expr.induct)
apply simp+
apply (case_tac binop)
apply simp+
apply (simp add: pushST_def split_beta)
apply simp+
done
```

```
lemma length_compTpExpr: "length (mt_of (compTpExpr jmb G ex sttp)) = length (compExpr
jmb ex)"
by (rule length_compTpExpr_Exprs [THEN conjunct1 [THEN spec]])

lemma length_compTpExprs: "length (mt_of (compTpExprs jmb G exs sttp)) = length (compExprs
jmb exs)"
by (rule length_compTpExpr_Exprs [THEN conjunct2 [THEN spec]])

lemma length_compTpStmt [rule_format]: "
  (∀ sttp. (length (mt_of (compTpStmt jmb G s sttp)) = length (compStmt jmb s)))"
apply (rule stmt.induct)
apply (simp add: length_compTpExpr)+
done


lemma length_compTpInit: "length (mt_of (compTpInit jmb lv sttp)) = length (compInit
jmb lv)"
by (simp add: compTpInit_def compInit_def split_beta)

lemma length_compTpInitLvars [rule_format]:
  "∀ sttp. length (mt_of (compTpInitLvars jmb lvars sttp)) = length (compInitLvars jmb
lvars)"
by (induct lvars,  (simp add: compInitLvars_def length_compTpInit split_beta)+)
```

4.24.9 Correspondence bytecode - method types

```
abbreviation (input)
  ST_of :: "state_type ⇒ opstack_type"
  where "ST_of == fst"

abbreviation (input)
  LT_of :: "state_type ⇒ locvars_type"
  where "LT_of == snd"

lemma states_lower:
  "⟦ OK (Some (ST, LT)) ∈ states cG mxs mxr; length ST ≤ mxs⟧
  ⟹ OK (Some (ST, LT)) ∈ states cG (length ST) mxr"
apply (simp add: states_def JVMType.sl_def)
apply (simp add: Product.esl_def stk_esl_def reg_sl_def upto_esl_def Listn.sl_def Err.sl_def
  JType.esl_def)
apply (simp add: Err.esl_def Err.le_def Listn.le_def)
apply (simp add: Product.le_def Product.sup_def Err.sup_def)
apply (simp add: Opt.esl_def Listn.sup_def)
apply clarify
apply auto
done

lemma check_type_lower:
  "⟦ check_type cG mxs mxr (OK (Some (ST, LT))); length ST ≤ mxs⟧
  ⟹check_type cG (length ST) mxr (OK (Some (ST, LT)))"
by (simp add: check_type_def states_lower)
```

```
definition bc_mt_corresp :: "
  [bytecode, state_type ⇒ method_type × state_type, state_type, jvm_prog, ty, nat, p_count]
  ⇒ bool" where

  "bc_mt_corresp bc f sttp0 cG rT mxr idx ==
  let (mt, sttp) = f sttp0 in
  (length bc = length mt ∧
    ((check_type cG (length (ST_of sttp0)) mxr (OK (Some sttp0))) ⟶
    (∀  mxs.
     mxs = max_ssize (mt@[Some sttp]) ⟶
     (∀ pc. pc < idx ⟶
      wt_instr_altern (bc ! pc) cG rT (mt@[Some sttp]) mxs mxr (length mt + 1) empty_et
pc)
     ∧
     check_type cG mxs mxr (OK ((mt@[Some sttp]) ! idx)))))"


lemma bc_mt_corresp_comb: "
  ⟦ bc' = (bc1@bc2); l' = (length bc');
  bc_mt_corresp bc1 f1 sttp0 cG rT mxr (length bc1);
  bc_mt_corresp bc2 f2 (sttp_of (f1 sttp0)) cG rT mxr (length bc2);
  start_sttp_resp f2⟧
⟹ bc_mt_corresp bc' (f1 □ f2) sttp0 cG rT mxr l'"
apply (subgoal_tac "∃ mt1 sttp1. (f1 sttp0) = (mt1, sttp1)", (erule exE)+)
apply (subgoal_tac "∃ mt2 sttp2. (f2 sttp1) = (mt2, sttp2)", (erule exE)+)


apply (simp only: start_sttp_resp_def)
apply (erule disjE)

apply (simp add: bc_mt_corresp_def comb_nil_def start_sttp_resp_cons_def)
apply (erule conjE)+
apply (intro strip)
apply simp


apply (simp add: bc_mt_corresp_def comb_def start_sttp_resp_cons_def del: all_simps)
apply (intro strip)
apply (erule conjE)+
apply (drule mp, assumption)
apply (subgoal_tac "check_type cG (length (fst sttp1)) mxr (OK (Some sttp1))")
apply (erule conjE)+
apply (drule mp, assumption)
apply (erule conjE)+

apply (rule conjI)


apply (drule_tac x=sttp1 in spec, simp)
apply (erule exE)
apply (intro strip)
apply (case_tac "pc < length mt1")
```

```
apply (drule spec, drule mp, simp)
apply simp
apply (rule_tac mt="mt1 @ [Some sttp1]" in wt_instr_prefix)
  apply assumption+ apply (rule HOL.refl)
  apply (simp (no_asm_simp))
  apply (simp (no_asm_simp) add: max_ssize_def)
  apply (simp add: max_of_list_def max_ac)
  apply arith
  apply (simp (no_asm_simp))+


apply (rule_tac bc=bc2 and mt=mt2 and bc_post="[]" and mt_post="[Some sttp2]"
  and mxr=mxr
  in wt_instr_offset)
apply simp
apply (simp (no_asm_simp))+
apply simp
apply (simp add: max_ssize_def max_of_list_append) apply (simp (no_asm_simp))


apply (subgoal_tac "((mt2 @ [Some sttp2]) ! length bc2) = Some sttp2")
apply (simp only:)
apply (rule check_type_mono) apply assumption
apply (simp (no_asm_simp)  add: max_ssize_def max_of_list_append max_ac)
apply (simp add: nth_append)

apply (erule conjE)+
apply (case_tac sttp1)
apply (simp add: check_type_def)
apply (rule states_lower, assumption)
apply (simp (no_asm_simp) add: max_ssize_def max_of_list_append)
apply (simp (no_asm_simp) add: max_of_list_def ssize_sto_def)
apply (simp (no_asm_simp))+
done


lemma bc_mt_corresp_zero [simp]: "⟦ length (mt_of (f sttp)) = length bc; start_sttp_resp
f⟧
  ⟹ bc_mt_corresp bc f sttp cG rT mxr 0"
apply (simp add: bc_mt_corresp_def start_sttp_resp_def split_beta)
apply (erule disjE)
apply (simp add: max_ssize_def max_of_list_def ssize_sto_def split: prod.split)
apply (intro strip)
apply (simp add: start_sttp_resp_cons_def split_beta)
apply (drule_tac x=sttp in spec, erule exE)
apply simp
apply (rule check_type_mono, assumption)
apply (simp add: max_ssize_def max_of_list_def ssize_sto_def split: prod.split)
done




definition mt_sttp_flatten :: "method_type × state_type ⇒ method_type" where
```

```
  "mt_sttp_flatten mt_sttp == (mt_of mt_sttp) @ [Some (sttp_of mt_sttp)]"


lemma mt_sttp_flatten_length [simp]: "n = (length (mt_of (f sttp)))
 ⟹ (mt_sttp_flatten (f sttp)) ! n = Some (sttp_of (f sttp))"
by (simp add: mt_sttp_flatten_def)

lemma mt_sttp_flatten_comb: "(mt_sttp_flatten ((f1 □ f2) sttp)) =
  (mt_of (f1 sttp)) @ (mt_sttp_flatten (f2 (sttp_of (f1 sttp))))"
by (simp add: mt_sttp_flatten_def comb_def split_beta)

lemma mt_sttp_flatten_comb_length [simp]: "⟦ n1 = length (mt_of (f1 sttp)); n1 ≤ n ⟧
  ⟹ (mt_sttp_flatten ((f1 □ f2) sttp) ! n) = (mt_sttp_flatten (f2 (sttp_of (f1 sttp)))
! (n - n1))"
by (simp add: mt_sttp_flatten_comb nth_append)

lemma mt_sttp_flatten_comb_zero [simp]: "start_sttp_resp f
  ⟹ (mt_sttp_flatten (f sttp)) ! 0 = Some sttp"
apply (simp only: start_sttp_resp_def)
apply (erule disjE)
apply (simp add: comb_nil_def mt_sttp_flatten_def)
apply (simp add: start_sttp_resp_cons_def mt_sttp_flatten_def split_beta)
apply (drule_tac x=sttp in spec)
apply (erule exE)
apply simp
done



lemma int_outside_right: "0 ≤ (m::int) ⟹ m + (int n) = int ((nat m) + n)"
by simp

lemma int_outside_left: "0 ≤ (m::int) ⟹ (int n) + m = int (n + (nat m))"
by simp







lemma less_Suc [simp] : "n ≤ k ⟹ (k < Suc n) = (k = n)"
  by arith

lemmas check_type_simps = check_type_def states_def JVMType.sl_def
   Product.esl_def stk_esl_def reg_sl_def upto_esl_def Listn.sl_def Err.sl_def
  JType.esl_def Err.esl_def Err.le_def Listn.le_def Product.le_def Product.sup_def Err.sup_def
  Opt.esl_def Listn.sup_def


lemma check_type_push: "⟦
  is_class cG cname; check_type cG (length ST) mxr (OK (Some (ST, LT))) ⟧
  ⟹ check_type cG (Suc (length ST)) mxr (OK (Some (Class cname # ST, LT)))"
```

```
apply (simp add: check_type_simps)
apply clarify
apply (rule_tac x="Suc (length ST)" in exI)
apply simp+
done

lemma bc_mt_corresp_New: "⟦is_class cG cname ⟧
  ⟹ bc_mt_corresp [New cname] (pushST [Class cname]) (ST, LT) cG rT mxr (Suc 0)"
apply (simp add: bc_mt_corresp_def pushST_def wt_instr_altern_def
    max_ssize_def max_of_list_def ssize_sto_def eff_def norm_eff_def min_max.sup_absorb2)
apply (intro strip)
apply (rule conjI)
apply (rule check_type_mono, assumption, simp)
apply (simp add: check_type_push)
done

lemma bc_mt_corresp_Pop: "
  bc_mt_corresp [Pop] (popST (Suc 0)) (T # ST, LT) cG rT mxr (Suc 0)"
  apply (simp add: bc_mt_corresp_def popST_def wt_instr_altern_def eff_def norm_eff_def)
  apply (simp add: max_ssize_def ssize_sto_def max_of_list_def)
  apply (simp add: check_type_simps min_max.sup_absorb1)
  apply clarify
  apply (rule_tac x="(length ST)" in exI)
  apply simp+
  done

lemma bc_mt_corresp_Checkcast: "⟦ is_class cG cname; sttp = (ST, LT);
  (∃rT STo. ST = RefT rT # STo) ⟧
  ⟹ bc_mt_corresp [Checkcast cname] (replST (Suc 0) (Class cname)) sttp cG rT mxr (Suc
0)"
  apply (erule exE)+
  apply (simp add: bc_mt_corresp_def replST_def wt_instr_altern_def eff_def norm_eff_def)
  apply (simp add: max_ssize_def max_of_list_def ssize_sto_def)
  apply (simp add: check_type_simps)
  apply clarify
  apply (rule_tac x="Suc (length STo)" in exI)
  apply simp+
  done


lemma bc_mt_corresp_LitPush: "⟦ typeof (λv. None) val = Some T ⟧
  ⟹ bc_mt_corresp [LitPush val] (pushST [T]) sttp cG rT mxr (Suc 0)"
apply (subgoal_tac "∃ ST LT. sttp= (ST, LT)", (erule exE)+)
  apply (simp add: bc_mt_corresp_def pushST_def wt_instr_altern_def
              max_ssize_def max_of_list_def ssize_sto_def eff_def norm_eff_def min_max.sup_abso
  apply (intro strip)
  apply (rule conjI)
  apply (rule check_type_mono, assumption, simp)
  apply (simp add: check_type_simps)
apply clarify
apply (rule_tac x="Suc (length ST)" in exI)
apply simp
apply (drule sym)
apply (case_tac val)
```

```
apply simp+
done


lemma bc_mt_corresp_LitPush_CT: "⟦ typeof (λv. None) val = Some T ∧ cG ⊢ T ≼ T';
  is_type cG T' ⟧
  ⟹ bc_mt_corresp [LitPush val] (pushST [T']) sttp cG rT mxr (Suc 0)"
apply (subgoal_tac "∃ ST LT. sttp= (ST, LT)", (erule exE)+)
  apply (simp add: bc_mt_corresp_def pushST_def wt_instr_altern_def
            max_ssize_def max_of_list_def ssize_sto_def eff_def norm_eff_def min_max.sup_absorb2)
  apply (intro strip)
  apply (rule conjI)
  apply (rule check_type_mono, assumption, simp)
  apply (simp add: check_type_simps)
  apply (simp add: sup_state_Cons)
apply clarify
apply (rule_tac x="Suc (length ST)" in exI)
apply simp
apply simp+
done

declare not_Err_eq [iff del]

lemma bc_mt_corresp_Load: "⟦ i < length LT; LT ! i ≠ Err; mxr = length LT ⟧
  ⟹ bc_mt_corresp [Load i]
        (λ(ST, LT). pushST [ok_val (LT ! i)] (ST, LT)) (ST, LT) cG rT mxr (Suc 0)"
apply (simp add: bc_mt_corresp_def pushST_def wt_instr_altern_def
          max_ssize_def max_of_list_def ssize_sto_def eff_def norm_eff_def min_max.sup_absorb2)
  apply (intro strip)
  apply (rule conjI)
  apply (rule check_type_mono, assumption, simp)
apply (simp add: check_type_simps)
apply clarify
apply (rule_tac x="Suc (length ST)" in exI)
apply (simp (no_asm_simp))
  apply (simp only: err_def)
  apply (frule listE_nth_in) apply assumption
apply (subgoal_tac "LT ! i ∈ {x. ∃y∈types cG. x = OK y}")
apply (drule CollectD) apply (erule bexE)
apply (simp (no_asm_simp))
apply blast
apply blast
done


lemma bc_mt_corresp_Store_init: "⟦ i < length LT ⟧
  ⟹ bc_mt_corresp [Store i] (storeST i T) (T # ST, LT) cG rT mxr (Suc 0)"
 apply (simp add: bc_mt_corresp_def storeST_def wt_instr_altern_def eff_def norm_eff_def)
  apply (simp add: max_ssize_def  max_of_list_def)
  apply (simp add: ssize_sto_def)
  apply (intro strip)
apply (simp add: check_type_simps min_max.sup_absorb1)
apply clarify
apply (rule conjI)
```

```
apply (rule_tac x="(length ST)" in exI)
apply simp+
done


lemma bc_mt_corresp_Store: "⟦ i < length LT; cG  ⊢  LT[i := OK T] <=l LT ⟧
  ⟹ bc_mt_corresp [Store i] (popST (Suc 0)) (T # ST, LT) cG rT mxr (Suc 0)"
  apply (simp add: bc_mt_corresp_def popST_def wt_instr_altern_def eff_def norm_eff_def)
  apply (simp add: sup_state_conv)
  apply (simp add: max_ssize_def max_of_list_def ssize_sto_def)
 apply (intro strip)
apply (simp add: check_type_simps min_max.sup_absorb1)
apply clarify
apply (rule_tac x="(length ST)" in exI)
apply simp+
done


lemma bc_mt_corresp_Dup: "
  bc_mt_corresp [Dup] dupST (T # ST, LT) cG rT mxr (Suc 0)"
 apply (simp add: bc_mt_corresp_def dupST_def wt_instr_altern_def
           max_ssize_def max_of_list_def ssize_sto_def eff_def norm_eff_def min_max.sup_absor
  apply (intro strip)
  apply (rule conjI)
  apply (rule check_type_mono, assumption, simp)
apply (simp add: check_type_simps)
apply clarify
apply (rule_tac x="Suc (Suc (length ST))" in exI)
apply simp+
done

lemma bc_mt_corresp_Dup_x1: "
  bc_mt_corresp [Dup_x1] dup_x1ST (T1 # T2 # ST, LT) cG rT mxr (Suc 0)"
  apply (simp add: bc_mt_corresp_def dup_x1ST_def wt_instr_altern_def
            max_ssize_def max_of_list_def ssize_sto_def eff_def norm_eff_def min_max.sup_abso
  apply (intro strip)
  apply (rule conjI)
  apply (rule check_type_mono, assumption, simp)
apply (simp add: check_type_simps)
apply clarify
apply (rule_tac x="Suc (Suc (Suc (length ST)))" in exI)
apply simp+
done



lemma bc_mt_corresp_IAdd: "
  bc_mt_corresp [IAdd] (replST 2 (PrimT Integer))
        (PrimT Integer # PrimT Integer # ST, LT) cG rT mxr (Suc 0)"
  apply (simp add: bc_mt_corresp_def replST_def wt_instr_altern_def eff_def norm_eff_def)
  apply (simp add: max_ssize_def max_of_list_def ssize_sto_def)
  apply (simp add: check_type_simps min_max.sup_absorb1)
  apply clarify
  apply (rule_tac x="Suc (length ST)" in exI)
```

```
  apply simp+
  done

lemma bc_mt_corresp_Getfield: "⟦ wf_prog wf_mb G;
  field (G, C) vname = Some (cname, T);  is_class G C ⟧
  ⟹ bc_mt_corresp [Getfield vname cname]
        (replST (Suc 0) (snd (the (field (G, cname) vname))))
        (Class C # ST, LT) (comp G) rT mxr (Suc 0)"
  apply (frule wf_prog_ws_prog [THEN wf_subcls1])
  apply (frule field_in_fd, assumption+)
  apply (frule widen_field, assumption+)
  apply (simp add: bc_mt_corresp_def replST_def wt_instr_altern_def eff_def norm_eff_def)
  apply (simp add: comp_field comp_subcls1 comp_widen comp_is_class)
  apply (simp add: max_ssize_def max_of_list_def ssize_sto_def)
  apply (intro strip)
apply (simp add: check_type_simps)
apply clarify
apply (rule_tac x="Suc (length ST)" in exI)
apply simp+
apply (simp only: comp_is_type)
apply (rule_tac C=cname in fields_is_type)
apply (simp add: TypeRel.field_def)
apply (drule JBasis.table_of_remap_SomeD)+
apply assumption+
apply (erule wf_prog_ws_prog)
apply assumption
done

lemma bc_mt_corresp_Putfield: "⟦ wf_prog wf_mb G;
  field (G, C) vname = Some (cname, Ta); G ⊢ T ≼ Ta; is_class G C ⟧
  ⟹ bc_mt_corresp [Putfield vname cname] (popST 2) (T # Class C # T # ST, LT)
           (comp G) rT mxr (Suc 0)"
  apply (frule wf_prog_ws_prog [THEN wf_subcls1])
  apply (frule field_in_fd, assumption+)
  apply (frule widen_field, assumption+)
  apply (simp add: bc_mt_corresp_def popST_def wt_instr_altern_def eff_def norm_eff_def)
  apply (simp add: comp_field comp_subcls1 comp_widen comp_is_class)
  apply (simp add: max_ssize_def max_of_list_def ssize_sto_def)

  apply (intro strip)
apply (simp add: check_type_simps min_max.sup_absorb1)
apply clarify
apply (rule_tac x="Suc (length ST)" in exI)
apply simp+
done



lemma Call_app: "⟦ wf_prog wf_mb G; is_class G cname;
STs = rev pTsa @ Class cname # ST;
max_spec G cname (mname, pTsa) = {((md, T), pTs')} ⟧
  ⟹ app (Invoke cname mname pTs') (comp G) (length (T # ST)) rT 0 empty_et (Some (STs,
LTs))"
  apply (subgoal_tac "(∃mD' rT' comp_b.
```

```
    method (comp G, cname) (mname, pTs') = Some (mD', rT', comp_b))")
  apply (simp add: comp_is_class)
  apply (rule_tac x=pTsa in exI)
  apply (rule_tac x="Class cname" in exI)
  apply (simp add: max_spec_preserves_length comp_is_class)
  apply (frule max_spec2mheads, (erule exE)+, (erule conjE)+)
  apply (simp add: split_paired_all comp_widen list_all2_def)
  apply (frule max_spec2mheads, (erule exE)+, (erule conjE)+)
  apply (rule exI)+
  apply (simp add: wf_prog_ws_prog [THEN comp_method])
  apply auto
  done


lemma bc_mt_corresp_Invoke: "⟦ wf_prog wf_mb G;
  max_spec G cname (mname, pTsa) = {((md, T), fpTs)};
  is_class G cname ⟧
 ⟹ bc_mt_corresp [Invoke cname mname fpTs] (replST (Suc (length pTsa)) T)
           (rev pTsa @ Class cname # ST, LT) (comp G) rT mxr (Suc 0)"
  apply (simp add: bc_mt_corresp_def wt_instr_altern_def eff_def norm_eff_def)
  apply (simp add: replST_def del: appInvoke)
  apply (intro strip)
  apply (rule conjI)

  — app
  apply (rule Call_app [THEN app_mono_mxs]) apply assumption+
    apply (rule HOL.refl) apply assumption
    apply (simp add: max_ssize_def max_of_list_elem ssize_sto_def)

  — <=s
  apply (frule max_spec2mheads, (erule exE)+, (erule conjE)+)
  apply (simp add: wf_prog_ws_prog [THEN comp_method])
  apply (simp add: max_spec_preserves_length [THEN sym])

  — check_type
apply (simp add: max_ssize_def ssize_sto_def)
apply (simp add: max_of_list_def)
apply (subgoal_tac "(max (length pTsa + length ST) (length ST)) = (length pTsa + length
ST)")
apply simp
apply (simp add: check_type_simps)
apply clarify
apply (rule_tac x="Suc (length ST)" in exI)
apply simp+
apply (simp only: comp_is_type)
apply (frule method_wf_mdecl) apply assumption apply assumption
apply (simp add: wf_mdecl_def wf_mhead_def)
apply (simp)
  done


lemma wt_instr_Ifcmpeq: "⟦Suc pc < max_pc;
  0 ≤ (int pc + i);  nat (int pc + i) < max_pc;
  (mt_sttp_flatten f ! pc = Some (ts#ts'#ST,LT)) ∧
```

```
  ((∃p. ts = PrimT p ∧ ts' = PrimT p) ∨ (∃r r'. ts = RefT r ∧ ts' = RefT r'));
  mt_sttp_flatten f ! Suc pc = Some (ST,LT);
  mt_sttp_flatten f ! nat (int pc + i) = Some (ST,LT);
  check_type (TranslComp.comp G) mxs mxr (OK (Some (ts # ts' # ST, LT))) ⟧
    ⟹  wt_instr_altern (Ifcmpeq i) (comp G) rT  (mt_sttp_flatten f) mxs mxr max_pc empty_et
pc"
by (simp  add: wt_instr_altern_def eff_def norm_eff_def)


lemma wt_instr_Goto: "⟦ 0 ≤ (int pc + i); nat (int pc + i) < max_pc;
  mt_sttp_flatten f ! nat (int pc + i) = (mt_sttp_flatten f ! pc);
  check_type (TranslComp.comp G) mxs mxr (OK (mt_sttp_flatten f ! pc)) ⟧
  ⟹ wt_instr_altern (Goto i) (comp G) rT  (mt_sttp_flatten f) mxs mxr max_pc empty_et
pc"
apply (case_tac "(mt_sttp_flatten f ! pc)")
apply (simp  add: wt_instr_altern_def eff_def norm_eff_def app_def xcpt_app_def)+
done


lemma bc_mt_corresp_comb_inside: "
  ⟦
  bc_mt_corresp bc' f' sttp0 cG rT mxr l1;
  bc' = (bc1@bc2@bc3); f'= (f1 □ f2 □ f3);
  l1 = (length bc1); l12 = (length (bc1@bc2));
  bc_mt_corresp bc2 f2 (sttp_of (f1 sttp0)) cG rT mxr (length bc2);
  length bc1 = length (mt_of (f1 sttp0));
  start_sttp_resp f2; start_sttp_resp f3⟧
⟹ bc_mt_corresp bc' f' sttp0 cG rT mxr l12"
apply (subgoal_tac "∃ mt1 sttp1. (f1 sttp0) = (mt1, sttp1)", (erule exE)+)
apply (subgoal_tac "∃ mt2 sttp2. (f2 sttp1) = (mt2, sttp2)", (erule exE)+)
apply (subgoal_tac "∃ mt3 sttp3. (f3 sttp2) = (mt3, sttp3)", (erule exE)+)


apply (simp only: start_sttp_resp_def)
apply (erule_tac Q="start_sttp_resp_cons f2" in disjE)

apply (simp add: bc_mt_corresp_def comb_nil_def start_sttp_resp_cons_def)


apply (simp add: bc_mt_corresp_def comb_def start_sttp_resp_cons_def)
apply (drule_tac x=sttp1 in spec, simp, erule exE)
apply (intro strip, (erule conjE)+)


apply (subgoal_tac "check_type cG (length (fst sttp1)) mxr (OK (Some sttp1))")
apply (subgoal_tac "check_type cG (max_ssize (mt2 @ [Some sttp2])) mxr (OK (Some sttp2))")
apply (subgoal_tac "check_type cG (max_ssize (mt1 @ mt2 @ mt3 @ [Some sttp3])) mxr
```

```
          (OK ((mt2 @ mt3 @ [Some sttp3]) ! length mt2))")
apply simp



apply (intro strip, (erule conjE)+)
apply (case_tac "pc < length mt1")


apply (drule spec, drule mp, assumption)
apply assumption



apply (erule_tac P="f3 = comb_nil" in disjE)


apply (subgoal_tac "mt3 = [] ∧ sttp2 = sttp3")  apply (erule conjE)+
apply (subgoal_tac "bc3=[]")

apply (rule_tac bc_pre=bc1 and bc=bc2 and bc_post=bc3
  and mt_pre=mt1 and mt=mt2 and mt_post="mt3@ [Some sttp3]"
  and mxs="(max_ssize (mt2 @ [(Some sttp2)]))"
  and max_pc="(Suc (length mt2))"
  in wt_instr_offset)
  apply simp
  apply (rule HOL.refl)+
  apply (simp (no_asm_simp))+

  apply (simp (no_asm_simp) add: max_ssize_def del: max_of_list_append)
    apply (rule max_of_list_sublist)
    apply (simp (no_asm_simp) only: set_append set.simps map.simps) apply blast
  apply (simp (no_asm_simp))
  apply simp
  apply (simp add: comb_nil_def)


apply (subgoal_tac "∃mt3_rest. (mt3 = Some sttp2 # mt3_rest)", erule exE)
apply (rule_tac bc_pre=bc1 and bc=bc2 and bc_post=bc3
  and mt_pre=mt1 and mt=mt2 and mt_post="mt3@ [Some sttp3]"
  and mxs="(max_ssize (mt2 @ [Some sttp2]))"
  and max_pc="(Suc (length mt2))"
  in wt_instr_offset)
apply (intro strip)
apply (rule_tac bc=bc2 and mt="(mt2 @ [Some sttp2])"
  and mxs="(max_ssize (mt2 @ [Some sttp2]))"
  and max_pc="(Suc (length mt2))"
  in wt_instr_prefix)




  apply simp
  apply (rule HOL.refl)
  apply (simp (no_asm_simp))+
```

```
  apply simp+

  apply (simp (no_asm_simp) add: max_ssize_def del: max_of_list_append)
  apply (rule max_of_list_sublist)
    apply (simp (no_asm_simp) only: set_append set.simps map.simps) apply blast
  apply (simp (no_asm_simp))

apply (drule_tac x=sttp2 in spec, simp)



apply simp

apply (erule_tac P="f3 = comb_nil" in disjE)


apply (subgoal_tac "mt3 = [] ∧ sttp2 = sttp3")  apply (erule conjE)+
apply simp
apply (rule check_type_mono, assumption)
apply (simp only: max_ssize_def) apply (rule max_of_list_sublist) apply (simp (no_asm_simp))
apply blast
  apply simp
  apply (simp add: comb_nil_def)



apply (subgoal_tac "∃mt3_rest. (mt3 = Some sttp2 # mt3_rest)", erule exE)
apply (simp (no_asm_simp) add: nth_append)
apply (erule conjE)+
apply (rule check_type_mono, assumption)
apply (simp only: max_ssize_def) apply (rule max_of_list_sublist) apply (simp (no_asm_simp))
apply blast
apply (drule_tac x=sttp2 in spec, simp)



apply (simp add: nth_append)


apply (simp add: nth_append)
apply (erule conjE)+
apply (case_tac "sttp1", simp)
apply (rule check_type_lower) apply assumption
apply (simp (no_asm_simp) add: max_ssize_def ssize_sto_def)
apply (simp (no_asm_simp) add: max_of_list_def)


apply (rule surj_pair)+
done



definition contracting :: "(state_type ⇒ method_type × state_type) ⇒ bool" where
  "contracting f == (∀ ST LT.
```

```
                    let (ST', LT') = sttp_of (f (ST, LT))
                    in (length ST' ≤ length ST ∧ set ST' ⊆ set ST  ∧
                        length LT' = length LT ∧ set LT' ⊆ set LT))"
```

lemma *set_drop_Suc [rule_format]: "∀ xs. set (drop (Suc n) xs) ⊆ set (drop n xs)"*
apply *(induct n)*
apply *simp*
apply *(intro strip)*
apply *(rule list.induct)*
apply *simp*
apply *simp* **apply** *blast*
apply *(intro strip)*
apply *(rule_tac*
  P="λ xs. set (drop (Suc (Suc n)) xs) ⊆ set (drop (Suc n) xs)" **in** *list.induct)*
apply *simp+*
done

lemma *set_drop_le [rule_format,simp]: "∀ n xs. n ≤ m ⟶ set (drop m xs) ⊆ set (drop n xs)"*
apply *(induct m)*
apply *simp*
apply *(intro strip)*
apply *(subgoal_tac "n ≤ m ∨ n = Suc m")*
apply *(erule disjE)*
apply *(frule_tac x=n* **in** *spec, drule_tac x=xs* **in** *spec, drule mp, assumption)*
apply *(rule set_drop_Suc [THEN subset_trans], assumption)*
apply *auto*
done

lemma *set_drop [simp] : "set (drop m xs) ⊆ set xs"*
apply *(rule_tac B="set (drop 0 xs)"* **in** *subset_trans)*
apply *(rule set_drop_le)*
apply *simp+*
done

lemma *contracting_popST [simp]: "contracting (popST n)"*
by *(simp add: contracting_def popST_def)*

lemma *contracting_nochangeST [simp]: "contracting nochangeST"*
by *(simp add: contracting_def nochangeST_def)*

lemma *check_type_contracting: "⟦ check_type cG mxs mxr (OK (Some sttp)); contracting f⟧*
  ⟹ check_type cG mxs mxr (OK (Some (sttp_of (f sttp))))"
apply *(subgoal_tac "∃ ST LT. sttp = (ST, LT)", (erule exE)+)*
apply *(simp add: check_type_simps contracting_def)*
apply *clarify*
apply *(drule_tac x=ST* **in** *spec, drule_tac x=LT* **in** *spec)*
apply *(case_tac "(sttp_of (f (ST, LT)))")*

```
apply simp
apply (erule conjE)+

apply (drule listE_set)+
apply (rule conjI)
apply (rule_tac x="length a" in exI) apply simp
apply (rule listI) apply simp apply blast
apply (rule listI) apply simp apply blast
apply auto
done




lemma bc_mt_corresp_comb_wt_instr: "
  ⟦ bc_mt_corresp bc' f' sttp0 cG rT mxr l1;
  bc' = (bc1@[inst]@bc3); f'= (f1 □ f2 □ f3);
  l1 = (length bc1);
  length bc1 = length (mt_of (f1 sttp0));
  length (mt_of (f2 (sttp_of (f1 sttp0)))) = 1;
  start_sttp_resp_cons f1; start_sttp_resp_cons f2; start_sttp_resp f3;

  check_type cG (max_ssize (mt_sttp_flatten (f' sttp0))) mxr
           (OK ((mt_sttp_flatten (f' sttp0)) ! (length bc1)))
  ⟶
  wt_instr_altern inst cG rT
      (mt_sttp_flatten (f' sttp0))
      (max_ssize (mt_sttp_flatten (f' sttp0)))
      mxr
      (Suc (length bc'))
      empty_et
      (length bc1);
  contracting f2
 ⟧
⟹ bc_mt_corresp bc' f' sttp0 cG rT mxr (length (bc1@[inst]))"
apply (subgoal_tac "∃ mt1 sttp1. (f1 sttp0) = (mt1, sttp1)", (erule exE)+)
apply (subgoal_tac "∃ mt2 sttp2. (f2 sttp1) = (mt2, sttp2)", (erule exE)+)
apply (subgoal_tac "∃ mt3 sttp3. (f3 sttp2) = (mt3, sttp3)", (erule exE)+)

apply (simp add: bc_mt_corresp_def comb_def start_sttp_resp_cons_def
  mt_sttp_flatten_def)

apply (intro strip, (erule conjE)+)
apply (drule mp, assumption)+ apply (erule conjE)+
apply (drule mp, assumption)
apply (rule conjI)


apply (intro strip)
apply (case_tac "pc < length mt1")


apply (drule spec, drule mp, assumption)
apply assumption
```

```
apply (subgoal_tac "pc = length mt1") prefer 2 apply arith
apply (simp only:)
apply (simp add: nth_append mt_sttp_flatten_def)


apply (simp add: start_sttp_resp_def)
apply (drule_tac x="sttp0" in spec, simp, erule exE)
apply (drule_tac x="sttp1" in spec, simp, erule exE)

apply (subgoal_tac "check_type cG (max_ssize (mt1 @ mt2 @ mt3 @ [Some sttp3])) mxr
                   (OK (Some (sttp_of (f2 sttp1))))")

apply (simp only:)

apply (erule disjE)

apply (subgoal_tac "((mt1 @ mt2 @ mt3 @ [Some sttp3]) ! Suc (length mt1)) = (Some (snd
(f2 sttp1)))")apply (subgoal_tac "mt3 = [] ∧ sttp2 = sttp3")  apply (erule conjE)+
apply (simp add: nth_append)
apply (simp add: comb_nil_def)
apply (simp add: nth_append comb_nil_def)


apply (simp add: start_sttp_resp_cons_def)
apply (drule_tac x="sttp2" in spec, simp, erule exE)
apply (simp  add: nth_append)


apply (rule check_type_contracting)
apply (subgoal_tac "((mt1 @ mt2 @ mt3 @ [Some sttp3]) ! length mt1) = (Some sttp1)")
apply (simp add: nth_append)
apply (simp add: nth_append)

apply assumption


apply (rule surj_pair)+
done


lemma compTpExpr_LT_ST_rewr [simp]: "⟦
  wf_java_prog G;
  wf_java_mdecl G C ((mn, pTs), rT, (pns, lvars, blk, res));
  local_env G C (mn, pTs) pns lvars ⊢ ex :: T;
  is_inited_LT C pTs lvars LT⟧
⟹ sttp_of (compTpExpr (pns, lvars, blk, res) G ex (ST, LT)) = (T # ST, LT)"
apply (rule compTpExpr_LT_ST)
apply auto
done
```

```
lemma wt_method_compTpExpr_Exprs_corresp: "
  ⟦ jmb = (pns,lvars,blk,res);
  wf_prog wf_java_mdecl G;
  wf_java_mdecl G C ((mn, pTs), rT, jmb);
  E = (local_env G C (mn, pTs) pns lvars)⟧
⟹
 (∀ ST LT T bc' f'.
  E ⊢ ex :: T ⟶
  (is_inited_LT C pTs lvars LT) ⟶
  bc' = (compExpr jmb ex) ⟶
  f' = (compTpExpr jmb G ex)
  ⟶ bc_mt_corresp bc' f' (ST, LT) (comp G) rT (length LT) (length bc'))
 ∧
 (∀ ST LT Ts.
  E ⊢ exs [::] Ts ⟶
  (is_inited_LT C pTs lvars LT)
  ⟶ bc_mt_corresp (compExprs jmb exs) (compTpExprs jmb G exs) (ST, LT) (comp G) rT (length
LT) (length (compExprs jmb exs)))"

apply (rule  expr.induct)



apply (intro allI impI)
apply (simp only:)
apply (drule NewC_invers)
apply (simp (no_asm_use))
apply (rule bc_mt_corresp_New)
apply (simp add: comp_is_class)


apply (intro allI impI)
apply (simp only:)
apply (drule Cast_invers)
apply clarify
apply (simp (no_asm_use))
apply (rule bc_mt_corresp_comb) apply (rule HOL.refl, simp (no_asm_simp), blast)
apply (simp (no_asm_simp), rule bc_mt_corresp_Checkcast)
apply (simp add: comp_is_class)
apply (simp only: compTpExpr_LT_ST)
apply (drule cast_RefT)
apply blast
apply (simp add: start_sttp_resp_def)


apply (intro allI impI)
apply (simp only:)
apply (drule Lit_invers)

apply simp
```

```
apply (rule bc_mt_corresp_LitPush)
   apply assumption




apply (intro allI impI)
apply (simp (no_asm_simp) only:)
apply (drule BinOp_invers, erule exE, (erule conjE)+)
apply (case_tac binop)
apply (simp (no_asm_simp))


apply (subgoal_tac "bc_mt_corresp bc' f' (ST, LT) (comp G) rT (length LT) 0")
prefer 2
  apply (rule bc_mt_corresp_zero) apply (simp add: length_compTpExpr)
  apply (simp (no_asm_simp))

apply (drule_tac ?bc1.0="[]" and ?bc2.0 = "compExpr jmb expr1"
  and ?f1.0=comb_nil and ?f2.0 = "compTpExpr jmb G expr1"
  in bc_mt_corresp_comb_inside)
  apply (simp (no_asm_simp))+
  apply blast
  apply (simp (no_asm_simp) add: length_compTpExpr)+

apply (drule_tac ?bc2.0 = "compExpr jmb expr2" and ?f2.0 = "compTpExpr jmb G expr2"
  in bc_mt_corresp_comb_inside)
  apply (simp (no_asm_simp))+
  apply (simp only: compTpExpr_LT_ST)
  apply (simp (no_asm_simp) add: length_compTpExpr)
  apply (simp (no_asm_simp))
  apply (simp (no_asm_simp))

apply (drule_tac ?bc1.0 = "compExpr jmb expr1 @ compExpr jmb expr2"
    and inst = "Ifcmpeq 3" and ?bc3.0 = "[LitPush (Bool False),Goto 2, LitPush (Bool
True)]"
  and ?f1.0="compTpExpr jmb G expr1 □ compTpExpr jmb G expr2"
  and ?f2.0="popST 2" and ?f3.0="pushST [PrimT Boolean] □ popST 1 □ pushST [PrimT Boolean]"
  in bc_mt_corresp_comb_wt_instr)
  apply (simp (no_asm_simp) add: length_compTpExpr)+


  apply (intro strip)
  apply (simp (no_asm_simp) add: wt_instr_altern_def length_compTpExpr eff_def)
  apply (simp (no_asm_simp) add: norm_eff_def)
  apply (simp (no_asm_simp) only: int_outside_left nat_int)
  apply (simp (no_asm_simp) add: length_compTpExpr)
  apply (simp only: compTpExpr_LT_ST)+
  apply (simp add: eff_def norm_eff_def popST_def pushST_def mt_sttp_flatten_def)
  apply (case_tac Ta) apply (simp (no_asm_simp)) apply (simp (no_asm_simp))
  apply (rule contracting_popST)

apply (drule_tac ?bc1.0 = "compExpr jmb expr1 @ compExpr jmb expr2 @ [Ifcmpeq 3]"
  and ?bc2.0 = "[LitPush (Bool False)]"
```

```
  and ?bc3.0 = "[Goto 2, LitPush (Bool True)]"
  and ?f1.0 = "compTpExpr jmb G expr1 □ compTpExpr jmb G expr2 □ popST 2"
  and ?f2.0 = "pushST [PrimT Boolean]"
  and ?f3.0 = "popST (Suc 0) □ pushST [PrimT Boolean]"
  in bc_mt_corresp_comb_inside)
  apply (simp (no_asm_simp))+
  apply (simp add: compTpExpr_LT_ST_rewr popST_def)
  apply (rule_tac T="(PrimT Boolean)" in bc_mt_corresp_LitPush) apply (simp (no_asm_simp))
  apply (simp (no_asm_simp) add: length_compTpExpr)
  apply (simp (no_asm_simp))
  apply (simp (no_asm_simp) add: start_sttp_resp_def)


apply (drule_tac ?bc1.0 = "compExpr jmb expr1 @ compExpr jmb expr2 @ [Ifcmpeq 3, LitPush
(Bool False)]"
    and inst = "Goto 2" and ?bc3.0 = "[LitPush (Bool True)]"
  and ?f1.0="compTpExpr jmb G expr1 □ compTpExpr jmb G expr2 □ popST 2 □ pushST [PrimT
Boolean]"
  and ?f2.0="popST 1" and ?f3.0="pushST [PrimT Boolean]"
  in bc_mt_corresp_comb_wt_instr)
  apply (simp (no_asm_simp) add: length_compTpExpr)+


  apply (simp (no_asm_simp) add: wt_instr_altern_def length_compTpExpr)
  apply (simp (no_asm_simp) add: eff_def norm_eff_def)
  apply (simp (no_asm_simp) only: int_outside_right nat_int)
  apply (simp (no_asm_simp) add: length_compTpExpr)
  apply (simp only: compTpExpr_LT_ST)+
  apply (simp add: eff_def norm_eff_def popST_def pushST_def)
  apply (rule contracting_popST)

apply (drule_tac
  ?bc1.0 = "compExpr jmb expr1 @ compExpr jmb expr2 @ [Ifcmpeq 3, LitPush (Bool False),
Goto 2]"
  and ?bc2.0 = "[LitPush (Bool True)]"
  and ?bc3.0 = "[]"
  and ?f1.0 = "compTpExpr jmb G expr1 □ compTpExpr jmb G expr2 □ popST 2 □
                pushST [PrimT Boolean] □ popST (Suc 0)"
  and ?f2.0 = "pushST [PrimT Boolean]"
  and ?f3.0 = "comb_nil"
  in bc_mt_corresp_comb_inside)
  apply (simp (no_asm_simp))+
  apply (simp add: compTpExpr_LT_ST_rewr popST_def)
  apply (rule_tac T="(PrimT Boolean)" in bc_mt_corresp_LitPush) apply (simp (no_asm_simp))
  apply (simp (no_asm_simp) add: length_compTpExpr)
  apply (simp (no_asm_simp) add: start_sttp_resp_def)
  apply (simp (no_asm_simp))

apply simp


apply simp
apply (rule bc_mt_corresp_comb) apply (rule HOL.refl) apply simp apply blast
apply (rule bc_mt_corresp_comb, rule HOL.refl)
```

```
   apply (simp only: compTpExpr_LT_ST)
apply (simp only: compTpExpr_LT_ST) apply blast

apply (simp only: compTpExpr_LT_ST)
apply simp
apply (rule bc_mt_corresp_IAdd)
apply (simp (no_asm_simp) add: start_sttp_resp_def)
apply (simp (no_asm_simp) add: start_sttp_resp_def)



apply (intro allI impI)
apply (simp only:)
apply (drule LAcc_invers)
apply (frule wf_java_mdecl_length_pTs_pns)
apply clarify
apply (simp add: is_inited_LT_def)
apply (rule bc_mt_corresp_Load)
  apply (rule index_in_bounds) apply simp apply assumption
  apply (rule inited_LT_at_index_no_err)
  apply (rule index_in_bounds) apply simp apply assumption
apply (rule HOL.refl)



apply (intro allI impI)
apply (simp only:)
apply (drule LAss_invers, erule exE, (erule conjE)+)
apply (drule LAcc_invers)
apply (frule wf_java_mdecl_disjoint_varnames, simp add: disjoint_varnames_def)
apply (frule wf_java_mdecl_length_pTs_pns)
apply clarify
apply (simp (no_asm_use))
apply (rule bc_mt_corresp_comb) apply (rule HOL.refl, simp (no_asm_simp), blast)
apply (rule_tac ?bc1.0="[Dup]" and ?bc2.0="[Store (index (pns, lvars, blk, res) vname)]"

       and ?f1.0="dupST" and ?f2.0="popST (Suc 0)"
       in bc_mt_corresp_comb)
  apply (simp (no_asm_simp))+
  apply (rule bc_mt_corresp_Dup)
  apply (simp only: compTpExpr_LT_ST)
  apply (simp add: dupST_def is_inited_LT_def)
  apply (rule bc_mt_corresp_Store)
  apply (rule index_in_bounds)
    apply simp apply assumption
  apply (rule sup_loc_update_index, assumption+)
    apply simp apply assumption+
apply (simp add: start_sttp_resp_def)
apply (simp add: start_sttp_resp_def)


apply (intro allI impI)
apply (simp only:)
apply (drule FAcc_invers)
```

```
apply clarify
apply (simp (no_asm_use))
apply (rule bc_mt_corresp_comb) apply (rule HOL.refl, simp (no_asm_simp), blast)
  apply (simp (no_asm_simp))
  apply (rule bc_mt_corresp_Getfield) apply assumption+
     apply (fast intro: wt_class_expr_is_class)
apply (simp (no_asm_simp) add: start_sttp_resp_def)



apply (intro allI impI)
apply (simp only:)
apply (drule FAss_invers, erule exE, (erule conjE)+)
apply (drule FAcc_invers)
apply clarify
apply (simp (no_asm_use))
apply (rule bc_mt_corresp_comb) apply (rule HOL.refl) apply simp apply blast
  apply (simp only: compTpExpr_LT_ST)
apply (rule bc_mt_corresp_comb, (rule HOL.refl)+) apply blast
  apply (simp only: compTpExpr_LT_ST)
apply (rule_tac ?bc1.0="[Dup_x1]" and ?bc2.0="[Putfield vname cname]" in bc_mt_corresp_comb)

  apply (simp (no_asm_simp))+
apply (rule bc_mt_corresp_Dup_x1)
  apply (simp (no_asm_simp) add: dup_x1ST_def)
apply (rule bc_mt_corresp_Putfield) apply assumption+
     apply (fast intro: wt_class_expr_is_class)
apply (simp (no_asm_simp) add: start_sttp_resp_def)
apply (simp (no_asm_simp) add: start_sttp_resp_def)
apply (simp (no_asm_simp) add: start_sttp_resp_def)


apply (intro allI impI)
apply (simp only:)
apply (drule Call_invers)
apply clarify
apply (simp (no_asm_use))
apply (rule bc_mt_corresp_comb) apply (rule HOL.refl) apply simp apply blast
  apply (simp only: compTpExpr_LT_ST)
apply (rule bc_mt_corresp_comb, (rule HOL.refl)+) apply blast
  apply (simp only: compTpExprs_LT_ST)
  apply (simp (no_asm_simp))
apply (rule bc_mt_corresp_Invoke) apply assumption+
     apply (fast intro: wt_class_expr_is_class)
apply (simp (no_asm_simp) add: start_sttp_resp_def)
apply (rule start_sttp_resp_comb)
  apply (simp (no_asm_simp))
  apply (simp (no_asm_simp) add: start_sttp_resp_def)




apply (intro allI impI)
```

```
apply (drule Nil_invers)
apply simp


apply (intro allI impI)
apply (drule Cons_invers, (erule exE)+, (erule conjE)+)
apply clarify
apply (simp (no_asm_use))
apply (rule bc_mt_corresp_comb) apply (rule HOL.refl) apply simp apply blast
  apply (simp only: compTpExpr_LT_ST)
apply blast
apply simp

done


lemmas wt_method_compTpExpr_corresp [rule_format (no_asm)] =
  wt_method_compTpExpr_Exprs_corresp [THEN conjunct1]


lemma wt_method_compTpStmt_corresp [rule_format (no_asm)]: "
  ⟦ jmb = (pns,lvars,blk,res);
  wf_prog wf_java_mdecl G;
  wf_java_mdecl G C ((mn, pTs), rT, jmb);
  E = (local_env G C (mn, pTs) pns lvars)⟧
⟹
 (∀ ST LT T bc' f'.
  E ⊢ s√ ⟶
  (is_inited_LT C pTs lvars LT) ⟶
  bc' = (compStmt jmb s) ⟶
  f' = (compTpStmt jmb G s)
  ⟶ bc_mt_corresp bc' f' (ST, LT) (comp G) rT (length LT) (length bc'))"

apply (rule stmt.induct)


apply (intro allI impI)
apply simp


apply (intro allI impI)
apply (drule Expr_invers, erule exE)
apply (simp (no_asm_simp))
apply (rule bc_mt_corresp_comb) apply (rule HOL.refl, simp (no_asm_simp))
apply (rule wt_method_compTpExpr_corresp) apply assumption+
apply (simp add: compTpExpr_LT_ST [of _ pns lvars blk res])+
apply (rule bc_mt_corresp_Pop)
```

```
apply (simp add: start_sttp_resp_def)



apply (intro allI impI)
apply (drule Comp_invers)
apply clarify
apply (simp (no_asm_use))
apply (rule bc_mt_corresp_comb) apply (rule HOL.refl)
   apply (simp (no_asm_simp)) apply blast
apply (simp only: compTpStmt_LT_ST)
apply (simp (no_asm_simp))


apply (intro allI impI)
apply (simp (no_asm_simp) only:)
apply (drule Cond_invers, (erule conjE)+)
apply (simp (no_asm_simp))

apply (subgoal_tac "bc_mt_corresp bc' f' (ST, LT) (comp G) rT (length LT) 0")
prefer 2
apply (rule bc_mt_corresp_zero)
  apply (simp (no_asm_simp) add: length_compTpStmt length_compTpExpr)
  apply (simp (no_asm_simp))

apply (drule_tac ?bc1.0="[]" and ?bc2.0 = "[LitPush (Bool False)]"
  and ?bc3.0="compExpr jmb expr @ Ifcmpeq (2 + int (length (compStmt jmb stmt1))) #
          compStmt jmb stmt1 @ Goto (1 + int (length (compStmt jmb stmt2))) #
          compStmt jmb stmt2"
  and ?f1.0=comb_nil and ?f2.0 = "pushST [PrimT Boolean]"
  and ?f3.0="compTpExpr jmb G expr □ popST 2 □ compTpStmt jmb G stmt1 □
          nochangeST □ compTpStmt jmb G stmt2"
  in bc_mt_corresp_comb_inside)
  apply (simp (no_asm_simp))+
  apply (rule_tac T="(PrimT Boolean)" in bc_mt_corresp_LitPush)
  apply (simp (no_asm_simp) add: start_sttp_resp_def)+

apply (drule_tac ?bc1.0="[LitPush (Bool False)]" and ?bc2.0 = "compExpr jmb expr"
  and ?bc3.0="Ifcmpeq (2 + int (length (compStmt jmb stmt1))) #
          compStmt jmb stmt1 @ Goto (1 + int (length (compStmt jmb stmt2))) #
          compStmt jmb stmt2"
  and ?f1.0="pushST [PrimT Boolean]" and ?f2.0 = "compTpExpr jmb G expr"
  and ?f3.0="popST 2 □ compTpStmt jmb G stmt1 □
          nochangeST □ compTpStmt jmb G stmt2"
  in bc_mt_corresp_comb_inside)
  apply (simp (no_asm_simp))+
  apply (simp (no_asm_simp)  add: pushST_def)
  apply (rule  wt_method_compTpExpr_corresp) apply assumption+
      apply (simp (no_asm_simp))+


apply (drule_tac ?bc1.0 = "[LitPush (Bool False)] @ compExpr jmb expr"
    and inst = "Ifcmpeq (2 + int (length (compStmt jmb stmt1)))"
  and ?bc3.0 = "compStmt jmb stmt1 @ Goto (1 + int (length (compStmt jmb stmt2))) #
```

```
          compStmt jmb stmt2"
  and ?f1.0="pushST [PrimT Boolean] □ compTpExpr jmb G expr" and ?f2.0 = "popST 2"
  and ?f3.0="compTpStmt jmb G stmt1 □ nochangeST □ compTpStmt jmb G stmt2"
  in bc_mt_corresp_comb_wt_instr)
  apply (simp (no_asm_simp) add: length_compTpExpr)+
  apply (simp (no_asm_simp) add: start_sttp_resp_comb)


  apply (intro strip)
  apply (rule_tac ts="PrimT Boolean" and ts'="PrimT Boolean"
    and ST=ST and LT=LT
    in wt_instr_Ifcmpeq)
  apply (simp (no_asm_simp))
  apply (simp (no_asm_simp) only: int_outside_right nat_int, simp (no_asm_simp))
  apply (simp (no_asm_simp) only: int_outside_right nat_int, simp (no_asm_simp))

  apply (simp add: length_compTpExpr pushST_def)
  apply (simp only: compTpExpr_LT_ST)

  apply (simp add: length_compTpExpr pushST_def)
  apply (simp add: popST_def start_sttp_resp_comb)

  apply (simp (no_asm_simp) only: int_outside_right nat_int, simp (no_asm_simp))
  apply (simp add: length_compTpExpr pushST_def)
  apply (simp add: popST_def start_sttp_resp_comb length_compTpStmt)
  apply (simp only: compTpStmt_LT_ST)
  apply (simp add: nochangeST_def)

  apply (subgoal_tac "
    (mt_sttp_flatten (f' (ST, LT)) ! length ([LitPush (Bool False)] @ compExpr jmb expr))
=
    (Some (PrimT Boolean # PrimT Boolean # ST, LT))")
  apply (simp only:)
  apply (simp (no_asm_simp)) apply (rule trans, rule mt_sttp_flatten_comb_length)
    apply (rule HOL.refl) apply (simp (no_asm_simp) add: length_compTpExpr)
    apply (simp (no_asm_simp) add: length_compTpExpr pushST_def)
    apply (simp only: compTpExpr_LT_ST_rewr)

  apply (rule contracting_popST)

apply (drule_tac
  ?bc1.0="[LitPush (Bool False)] @ compExpr jmb expr @
          [Ifcmpeq (2 + int (length (compStmt jmb stmt1)))] "
  and ?bc2.0 = "compStmt jmb stmt1"
  and ?bc3.0="Goto (1 + int (length (compStmt jmb stmt2))) # compStmt jmb stmt2"
  and ?f1.0="pushST [PrimT Boolean] □ compTpExpr jmb G expr □ popST 2"
  and ?f2.0 = "compTpStmt jmb G stmt1"
  and ?f3.0="nochangeST □ compTpStmt jmb G stmt2"
  in bc_mt_corresp_comb_inside)
  apply (simp (no_asm_simp))+
  apply (simp (no_asm_simp) add: pushST_def popST_def compTpExpr_LT_ST)
  apply (simp only: compTpExpr_LT_ST)
  apply (simp (no_asm_simp))
  apply (simp (no_asm_simp) add: length_compTpExpr)+
```

```
apply (drule_tac ?bc1.0 = "[LitPush (Bool False)] @ compExpr jmb expr @ [Ifcmpeq (2 +
int (length (compStmt jmb stmt1)))] @ compStmt jmb stmt1"
    and inst = "Goto (1 + int (length (compStmt jmb stmt2)))"
  and ?bc3.0 = "compStmt jmb stmt2"
  and ?f1.0="pushST [PrimT Boolean] □ compTpExpr jmb G expr □ popST 2 □
             compTpStmt jmb G stmt1"
  and ?f2.0 = "nochangeST"
  and ?f3.0="compTpStmt jmb G stmt2"
  in bc_mt_corresp_comb_wt_instr)
  apply (simp (no_asm_simp) add: length_compTpExpr length_compTpStmt)+
  apply (intro strip)
  apply (rule wt_instr_Goto)
  apply (simp (no_asm_simp) only: int_outside_right nat_int, simp (no_asm_simp))
  apply (simp (no_asm_simp) only: int_outside_right nat_int, simp (no_asm_simp))

  apply (simp (no_asm_simp) add: length_compTpExpr length_compTpStmt)
  apply (simp (no_asm_simp) add: pushST_def popST_def nochangeST_def)
  apply (simp only: compTpExpr_LT_ST compTpStmt_LT_ST)
  apply (simp (no_asm_simp) add: pushST_def popST_def nochangeST_def)
  apply (simp only: compTpExpr_LT_ST compTpStmt_LT_ST)
  apply (simp only:)
  apply (simp add: length_compTpExpr length_compTpStmt)
  apply (rule contracting_nochangeST)


apply (drule_tac
  ?bc1.0= "[LitPush (Bool False)] @ compExpr jmb expr @
           [Ifcmpeq (2 + int (length (compStmt jmb stmt1)))] @
           compStmt jmb stmt1 @ [Goto (1 + int (length (compStmt jmb stmt2)))]"
  and ?bc2.0 = "compStmt jmb stmt2"
  and ?bc3.0="[]"
  and ?f1.0="pushST [PrimT Boolean] □ compTpExpr jmb G expr □ popST 2 □
             compTpStmt jmb G stmt1 □ nochangeST"
  and ?f2.0 = "compTpStmt jmb G stmt2"
  and ?f3.0="comb_nil"
  in bc_mt_corresp_comb_inside)
  apply (simp (no_asm_simp))+
  apply (simp (no_asm_simp) add: pushST_def popST_def nochangeST_def compTpExpr_LT_ST)
  apply (simp only: compTpExpr_LT_ST)
  apply (simp (no_asm_simp))
  apply (simp only: compTpStmt_LT_ST)
  apply (simp (no_asm_simp) add: length_compTpExpr length_compTpStmt)+

apply simp



apply (intro allI impI)
apply (simp (no_asm_simp) only:)
apply (drule Loop_invers, (erule conjE)+)
apply (simp (no_asm_simp))
```

```
apply (subgoal_tac "bc_mt_corresp bc' f' (ST, LT) (comp G) rT (length LT) 0")
prefer 2
apply (rule bc_mt_corresp_zero)
  apply (simp (no_asm_simp) add: length_compTpStmt length_compTpExpr)
  apply (simp (no_asm_simp))

apply (drule_tac ?bc1.0="[]" and ?bc2.0 = "[LitPush (Bool False)]"
  and ?bc3.0="compExpr jmb expr @ Ifcmpeq (2 + int (length (compStmt jmb stmt))) #
            compStmt jmb stmt @
            [Goto (-2 + (- int (length (compStmt jmb stmt)) - int (length (compExpr jmb
expr))))]"
  and ?f1.0=comb_nil and ?f2.0 = "pushST [PrimT Boolean]"
  and ?f3.0="compTpExpr jmb G expr □ popST 2 □ compTpStmt jmb G stmt □ nochangeST"
  in bc_mt_corresp_comb_inside)
  apply (simp (no_asm_simp))+
  apply (rule_tac T="(PrimT Boolean)" in bc_mt_corresp_LitPush)
  apply (simp (no_asm_simp) add: start_sttp_resp_def)+

apply (drule_tac ?bc1.0="[LitPush (Bool False)]" and ?bc2.0 = "compExpr jmb expr"
  and ?bc3.0="Ifcmpeq (2 + int (length (compStmt jmb stmt))) #
            compStmt jmb stmt @
            [Goto (-2 + (- int (length (compStmt jmb stmt)) - int (length (compExpr jmb
expr))))]"
  and ?f1.0="pushST [PrimT Boolean]" and ?f2.0 = "compTpExpr jmb G expr"
  and ?f3.0="popST 2 □ compTpStmt jmb G stmt □ nochangeST"
  in bc_mt_corresp_comb_inside)
  apply (simp (no_asm_simp))+
  apply (simp (no_asm_simp)  add: pushST_def)
  apply (rule  wt_method_compTpExpr_corresp) apply assumption+
      apply (simp (no_asm_simp))+


apply (drule_tac ?bc1.0 = "[LitPush (Bool False)] @ compExpr jmb expr"
    and inst = "Ifcmpeq (2 + int (length (compStmt jmb stmt)))"
  and ?bc3.0 = "compStmt jmb stmt @
      [Goto (-2 + (- int (length (compStmt jmb stmt)) - int (length (compExpr jmb expr))))]"
  and ?f1.0="pushST [PrimT Boolean] □ compTpExpr jmb G expr" and ?f2.0 = "popST 2"
  and ?f3.0="compTpStmt jmb G stmt □ nochangeST"
  in bc_mt_corresp_comb_wt_instr)
  apply (simp (no_asm_simp) add: length_compTpExpr)+
  apply (simp (no_asm_simp) add: start_sttp_resp_comb)


  apply (intro strip)
  apply (rule_tac ts="PrimT Boolean" and ts'="PrimT Boolean"
    and ST=ST and LT=LT
    in wt_instr_Ifcmpeq)
  apply (simp (no_asm_simp))
  apply (simp (no_asm_simp) only: int_outside_right nat_int, simp (no_asm_simp))
  apply (simp (no_asm_simp) only: int_outside_right nat_int, simp (no_asm_simp))

  apply (simp add: length_compTpExpr pushST_def)
  apply (simp only: compTpExpr_LT_ST)
```

```
  apply (simp add: length_compTpExpr pushST_def)
  apply (simp add: popST_def start_sttp_resp_comb)

  apply (simp (no_asm_simp) only: int_outside_right nat_int, simp (no_asm_simp))
  apply (simp add: length_compTpExpr pushST_def)
  apply (simp add: popST_def start_sttp_resp_comb length_compTpStmt)
  apply (simp only: compTpStmt_LT_ST)
  apply (simp add: nochangeST_def)

  apply (subgoal_tac "
    (mt_sttp_flatten (f' (ST, LT)) ! length ([LitPush (Bool False)] @ compExpr jmb expr))
=
    (Some (PrimT Boolean # PrimT Boolean # ST, LT))")
  apply (simp only:)
  apply (simp (no_asm_simp)) apply (rule trans, rule mt_sttp_flatten_comb_length)
    apply (rule HOL.refl) apply (simp (no_asm_simp) add: length_compTpExpr)
    apply (simp (no_asm_simp) add: length_compTpExpr pushST_def)
    apply (simp only: compTpExpr_LT_ST_rewr)

  apply (rule contracting_popST)

apply (drule_tac
  ?bc1.0="[LitPush (Bool False)] @ compExpr jmb expr @
          [Ifcmpeq (2 + int (length (compStmt jmb stmt)))] "
  and ?bc2.0 = "compStmt jmb stmt"
  and ?bc3.0="[Goto (-2 + (- int (length (compStmt jmb stmt)) - int (length (compExpr
jmb expr))))]"
  and ?f1.0="pushST [PrimT Boolean] □ compTpExpr jmb G expr □ popST 2"
  and ?f2.0 = "compTpStmt jmb G stmt"
  and ?f3.0="nochangeST"
  in bc_mt_corresp_comb_inside)
  apply (simp (no_asm_simp))+
  apply (simp (no_asm_simp) add: pushST_def popST_def compTpExpr_LT_ST)
  apply (simp only: compTpExpr_LT_ST)
  apply (simp (no_asm_simp))
  apply (simp (no_asm_simp) add: length_compTpExpr)+


apply (drule_tac ?bc1.0 = "[LitPush (Bool False)] @ compExpr jmb expr @ [Ifcmpeq (2 +
int (length (compStmt jmb stmt)))] @ compStmt jmb stmt"
    and inst = "Goto (-2 + (- int (length (compStmt jmb stmt)) - int (length (compExpr
jmb expr))))"
  and ?bc3.0 = "[]"
  and ?f1.0="pushST [PrimT Boolean] □ compTpExpr jmb G expr □ popST 2 □
            compTpStmt jmb G stmt"
  and ?f2.0 = "nochangeST"
  and ?f3.0="comb_nil"
  in bc_mt_corresp_comb_wt_instr)
  apply (simp (no_asm_simp) add: length_compTpExpr length_compTpStmt)+
  apply (intro strip)
  apply (rule wt_instr_Goto)
  apply arith
  apply arith
```

```
  apply (simp (no_asm_simp))
  apply (simp (no_asm_simp) add: length_compTpExpr length_compTpStmt)
  apply (simp (no_asm_simp) add: pushST_def popST_def nochangeST_def)
  apply (simp only: compTpExpr_LT_ST compTpStmt_LT_ST)
  apply (simp (no_asm_simp) add: length_compTpExpr length_compTpStmt)
  apply (simp only: compTpExpr_LT_ST compTpStmt_LT_ST)
  apply (simp (no_asm_simp) add: pushST_def popST_def nochangeST_def)
  apply (simp (no_asm_simp) add: length_compTpExpr length_compTpStmt)
  apply (simp only: compTpExpr_LT_ST compTpStmt_LT_ST)

  apply (simp add: length_compTpExpr length_compTpStmt)
    apply (simp add: pushST_def popST_def compTpExpr_LT_ST compTpStmt_LT_ST)
  apply (rule contracting_nochangeST)
apply simp

done
```

```
lemma wt_method_compTpInit_corresp: "⟦ jmb = (pns,lvars,blk,res);
  wf_java_mdecl G C ((mn, pTs), rT, jmb); mxr = length LT;
  length LT = (length pns) + (length lvars) + 1;  vn ∈ set (map fst lvars);
  bc = (compInit jmb (vn,ty)); f = (compTpInit jmb (vn,ty));
  is_type G ty ⟧
  ⟹ bc_mt_corresp bc f (ST, LT) (comp G) rT mxr (length bc)"
apply (simp add: compInit_def compTpInit_def split_beta)
apply (rule_tac ?bc1.0="[load_default_val ty]" and ?bc2.0="[Store (index jmb vn)]"
  in bc_mt_corresp_comb)
  apply simp+
  apply (simp add: load_default_val_def)
   apply (rule typeof_default_val [THEN exE])

  apply (rule bc_mt_corresp_LitPush_CT) apply assumption
    apply (simp add: comp_is_type)
apply (simp add: pushST_def)
  apply (rule bc_mt_corresp_Store_init)
  apply simp
  apply (rule index_length_lvars [THEN conjunct2])
apply auto
done
```

```
lemma wt_method_compTpInitLvars_corresp_aux [rule_format (no_asm)]: "
  ∀ lvars_pre lvars0 ST LT.
  jmb = (pns,lvars0,blk,res) ∧
  lvars0 = (lvars_pre @ lvars) ∧
  length LT = (length pns) + (length lvars0) + 1 ∧
  wf_java_mdecl G C ((mn, pTs), rT, jmb)
 ⟶ bc_mt_corresp (compInitLvars jmb lvars) (compTpInitLvars jmb lvars) (ST, LT) (comp
G) rT
       (length LT) (length (compInitLvars jmb lvars))"
```

```
apply (induct lvars)
apply (simp add: compInitLvars_def)

apply (intro strip, (erule conjE)+)
apply (subgoal_tac "∃ vn ty. a = (vn, ty)")
  prefer 2 apply (simp (no_asm_simp))
  apply ((erule exE)+, simp (no_asm_simp))
apply (drule_tac x="lvars_pre @ [a]" in spec)
apply (drule_tac x="lvars0" in spec)
apply (simp (no_asm_simp) add: compInitLvars_def)
apply (rule_tac ?bc1.0="compInit jmb a" and ?bc2.0="compInitLvars jmb lvars"
  in bc_mt_corresp_comb)
apply (simp (no_asm_simp) add: compInitLvars_def)+

apply (rule_tac vn=vn and ty=ty in wt_method_compTpInit_corresp)
apply assumption+
apply (simp (no_asm_simp))+
apply (simp add: wf_java_mdecl_def)
apply (simp add: compTpInit_def storeST_def pushST_def)
apply simp
done


lemma wt_method_compTpInitLvars_corresp: "⟦ jmb = (pns,lvars,blk,res);
  wf_java_mdecl G C ((mn, pTs), rT, jmb);
  length LT = (length pns) + (length lvars) + 1; mxr = (length LT);
  bc = (compInitLvars jmb lvars); f= (compTpInitLvars jmb lvars) ⟧
  ⟹ bc_mt_corresp bc f (ST, LT) (comp G) rT mxr (length bc)"
apply (simp only:)
apply (subgoal_tac "bc_mt_corresp (compInitLvars (pns, lvars, blk, res) lvars)
        (compTpInitLvars (pns, lvars, blk, res) lvars) (ST, LT) (TranslComp.comp G) rT
        (length LT) (length (compInitLvars (pns, lvars, blk, res) lvars))")
apply simp
apply (rule_tac lvars_pre="[]" in wt_method_compTpInitLvars_corresp_aux)
apply auto
done




lemma wt_method_comp_wo_return: "⟦ wf_prog wf_java_mdecl G;
  wf_java_mdecl G C ((mn, pTs), rT, jmb);
  bc = compInitLvars jmb lvars @ compStmt jmb blk @ compExpr jmb res;
  jmb = (pns,lvars,blk,res);
  f = (compTpInitLvars jmb lvars □ compTpStmt jmb G blk □ compTpExpr jmb G res);
  sttp = (start_ST, start_LT C pTs (length lvars));
  li = (length (inited_LT C pTs lvars))
  ⟧
⟹ bc_mt_corresp bc f sttp (comp G) rT  li (length bc)"
apply (subgoal_tac "∃ E. (E = (local_env G C (mn, pTs) pns lvars) ∧ E ⊢ blk √ ∧
                        (∃T. E⊢res::T ∧ G⊢T≼rT))")
   apply (erule exE, (erule conjE)+)
```

```
apply (simp only:)
apply (rule bc_mt_corresp_comb) apply (rule HOL.refl)+


apply (rule wt_method_compTpInitLvars_corresp)
   apply assumption+
   apply (simp only:)
   apply (simp (no_asm_simp) add: start_LT_def)
       apply (rule wf_java_mdecl_length_pTs_pns, assumption)
   apply (simp (no_asm_simp) only: start_LT_def)
   apply (simp (no_asm_simp) add: inited_LT_def)+

apply (rule bc_mt_corresp_comb) apply (rule HOL.refl)+
   apply (simp (no_asm_simp) add: compTpInitLvars_LT_ST)


apply (simp only: compTpInitLvars_LT_ST)
apply (subgoal_tac "(Suc (length pTs + length lvars)) = (length (inited_LT C pTs lvars))")
   prefer 2 apply (simp (no_asm_simp) add: inited_LT_def)
apply (simp only:)
apply (rule_tac s=blk in wt_method_compTpStmt_corresp)
   apply assumption+
   apply (simp only:)+
   apply (simp (no_asm_simp) add: is_inited_LT_def)
   apply (simp only:)+


apply (simp only: compTpInitLvars_LT_ST compTpStmt_LT_ST is_inited_LT_def)
apply (subgoal_tac "(Suc (length pTs + length lvars)) = (length (inited_LT C pTs lvars))")
   prefer 2 apply (simp (no_asm_simp) add: inited_LT_def)
apply (simp only:)
apply (rule_tac ex=res in wt_method_compTpExpr_corresp)
   apply assumption+
   apply (simp only:)+
   apply (simp (no_asm_simp) add: is_inited_LT_def)
   apply (simp only:)+


apply (simp add: start_sttp_resp_comb)+


apply (simp add: wf_java_mdecl_def local_env_def)
done





lemma check_type_start: "⟦ wf_mhead cG (mn, pTs) rT; is_class cG C⟧
  ⟹ check_type cG (length start_ST) (Suc (length pTs + mxl))
              (OK (Some (start_ST, start_LT C pTs mxl)))"
apply (simp add: check_type_def wf_mhead_def start_ST_def start_LT_def)
apply (simp add: check_type_simps)
```

```
apply (simp only: list_def)
apply (auto simp: err_def)
apply (subgoal_tac "set (replicate mxl Err) ⊆  {Err}")
apply blast
apply (rule subset_replicate)
done


lemma wt_method_comp_aux: "⟦   bc' = bc @ [Return]; f' = (f □ nochangeST);
  bc_mt_corresp bc f sttp0 cG rT (1+length pTs+mxl) (length bc);
  start_sttp_resp_cons f';
  sttp0 = (start_ST, start_LT C pTs mxl);
  mxs = max_ssize (mt_of (f' sttp0));
  wf_mhead cG (mn, pTs) rT; is_class cG C;
  sttp_of (f sttp0) = (T # ST, LT);

  check_type cG mxs (1+length pTs+mxl) (OK (Some (T # ST, LT))) ⟶
  wt_instr_altern Return cG rT (mt_of (f' sttp0)) mxs (1+length pTs+mxl)
         (Suc (length bc)) empty_et (length bc)
⟧
⟹ wt_method_altern cG C pTs rT mxs mxl bc' empty_et (mt_of (f' sttp0))"
apply (subgoal_tac "check_type cG (length start_ST) (Suc (length pTs + mxl))
              (OK (Some (start_ST, start_LT C pTs mxl)))")
apply (subgoal_tac "check_type cG mxs (1+length pTs+mxl) (OK (Some (T # ST, LT)))")

apply (simp add: wt_method_altern_def)


apply (rule conjI)
apply (simp add: bc_mt_corresp_def split_beta)


apply (rule conjI)
apply (simp add: bc_mt_corresp_def split_beta check_bounded_def)
apply (erule conjE)+
apply (intro strip)
apply (subgoal_tac "pc < (length bc) ∨ pc = length bc")
  apply (erule disjE)

    apply (subgoal_tac "(bc' ! pc) = (bc ! pc)")
    apply (simp add: wt_instr_altern_def eff_def)

    apply (simp add: nth_append)

    apply (subgoal_tac "(bc' ! pc) = Return")
    apply (simp add: wt_instr_altern_def)

    apply (simp add: nth_append)


apply arith


apply (rule conjI)
```

```
apply (simp add: wt_start_def start_sttp_resp_cons_def split_beta)
  apply (drule_tac x=sttp0 in spec) apply (erule exE)
  apply (simp add: mt_sttp_flatten_def start_ST_def start_LT_def)


apply (intro strip)
apply (subgoal_tac "pc < (length bc) ∨ pc = length bc")
apply (erule disjE)


apply (simp (no_asm_use) add: bc_mt_corresp_def mt_sttp_flatten_def split_beta)
apply (erule conjE)+
apply (drule mp, assumption)+
apply (erule conjE)+
apply (drule spec, drule mp, assumption)
apply (simp add: nth_append)
apply (simp (no_asm_simp) add: comb_def split_beta nochangeST_def)


apply (simp add: nth_append)


apply arith


apply (simp (no_asm_use) add: bc_mt_corresp_def split_beta)
apply (subgoal_tac "check_type cG (length (fst sttp0)) (Suc (length pTs + mxl))
         (OK (Some sttp0))")
apply ((erule conjE)+, drule mp, assumption)
apply (simp add: nth_append)
apply (simp (no_asm_simp) add: comb_def nochangeST_def split_beta)
apply (simp (no_asm_simp))

apply (rule check_type_start, assumption+)
done


lemma wt_instr_Return: "⟦fst f ! pc = Some (T # ST, LT); (G ⊢ T ≼ rT); pc < max_pc;
  check_type (TranslComp.comp G) mxs mxr (OK (Some (T # ST, LT)))
  ⟧
  ⟹ wt_instr_altern Return (comp G) rT  (mt_of f) mxs mxr max_pc empty_et pc"
  apply (case_tac "(mt_of f ! pc)")
  apply (simp  add: wt_instr_altern_def eff_def norm_eff_def app_def)+
  apply (drule sym)
  apply (simp add: comp_widen xcpt_app_def)
  done


theorem wt_method_comp: "
  ⟦ wf_java_prog G; (C, D, fds, mths) ∈ set G; jmdcl ∈ set mths;
  jmdcl = ((mn,pTs), rT, jmb);
  mt = (compTpMethod G C jmdcl);
  (mxs, mxl, bc, et) = mtd_mb (compMethod G C jmdcl) ⟧
  ⟹ wt_method (comp G) C pTs rT mxs mxl bc et mt"
```

```
apply (rule wt_method_altern_wt_method)

apply (subgoal_tac "wf_java_mdecl G C jmdcl")
apply (subgoal_tac "wf_mhead G (mn, pTs) rT")
apply (subgoal_tac "is_class G C")
apply (subgoal_tac "∀ jmb. ∃ pns lvars blk res. jmb = (pns, lvars, blk, res)")
   apply (drule_tac x=jmb in spec, (erule exE)+)
apply (subgoal_tac "∃ E. (E = (local_env G C (mn, pTs) pns lvars) ∧ E ⊢ blk √ ∧
                        (∃ T. E⊢res::T ∧ G⊢T≼rT))")
   apply (erule exE, (erule conjE)+)+
apply (simp add: compMethod_def compTpMethod_def split_beta)
apply (rule_tac T=T and LT="inited_LT C pTs lvars" and ST=start_ST in wt_method_comp_aux)


apply (simp only: append_assoc [THEN sym])


apply (simp only: comb_assoc [THEN sym])


apply (rule wt_method_comp_wo_return)
  apply assumption+
  apply (simp (no_asm_use) only: append_assoc)
  apply (rule HOL.refl)
  apply (simp (no_asm_simp))+
  apply (simp (no_asm_simp) add: inited_LT_def)


apply (simp add: start_sttp_resp_cons_comb_cons_r)+


apply (simp add: wf_mhead_def comp_is_type)
apply (simp add: comp_is_class)


apply (simp (no_asm_simp) add: compTpInitLvars_LT_ST compTpExpr_LT_ST compTpStmt_LT_ST
is_inited_LT_def)
apply (subgoal_tac "(snd (compTpInitLvars (pns, lvars, blk, res) lvars
                             (start_ST, start_LT C pTs (length lvars))))
                = (start_ST, inited_LT C pTs lvars)")
   prefer 2 apply (rule compTpInitLvars_LT_ST) apply (rule HOL.refl) apply assumption
apply (simp only:)
apply (subgoal_tac "(snd (compTpStmt (pns, lvars, blk, res) G blk
                       (start_ST, inited_LT C pTs lvars)))
                = (start_ST, inited_LT C pTs lvars)")
   prefer 2 apply (erule conjE)+
   apply (rule compTpStmt_LT_ST) apply (rule HOL.refl) apply assumption+
   apply (simp only:)+ apply (simp (no_asm_simp) add: is_inited_LT_def)
apply (simp only:)
apply (rule compTpExpr_LT_ST) apply (rule HOL.refl) apply assumption+
   apply (simp only:)+ apply (simp (no_asm_simp) add: is_inited_LT_def)
```

```
apply (intro strip)
apply (rule_tac T=T and ST=start_ST and LT="inited_LT C pTs lvars" in wt_instr_Return)
apply (simp (no_asm_simp) add: nth_append
  length_compTpInitLvars length_compTpStmt length_compTpExpr)
apply (simp only: compTpInitLvars_LT_ST compTpStmt_LT_ST compTpExpr_LT_ST
  nochangeST_def)
apply (simp only: is_inited_LT_def compTpStmt_LT_ST compTpExpr_LT_ST)
apply (simp (no_asm_simp))+
apply simp


apply (simp add: wf_java_mdecl_def local_env_def)


apply (simp only: split_paired_All, simp)


apply (blast intro: methd [THEN conjunct2])
apply (frule wf_prog_wf_mdecl, assumption+) apply (simp only:) apply (simp add: wf_mdecl_def)
apply (rule wf_java_prog_wf_java_mdecl, assumption+)
done


lemma comp_set_ms: "(C, D, fs, cms)∈set (comp G)
  ⟹ ∃ ms. (C, D, fs, ms) ∈set G   ∧ cms = map (compMethod G C) ms"
by (auto simp: comp_def compClass_def)
```

4.24.10 Main Theorem

```
theorem wt_prog_comp: "wf_java_prog G  ⟹ wt_jvm_prog (comp G) (compTp G)"
apply (simp add: wf_prog_def)

apply (subgoal_tac "wf_java_prog G") prefer 2 apply (simp add: wf_prog_def)
apply (simp (no_asm_simp) add: wf_prog_def wt_jvm_prog_def)
apply (simp add: comp_ws_prog)

apply (intro strip)
apply (subgoal_tac "∃ C D fs cms. c = (C, D, fs, cms)")
apply clarify
apply (frule comp_set_ms)
apply clarify
apply (drule bspec, assumption)
apply (rule conjI)


apply (case_tac "C = Object")
apply (simp add: wf_mrT_def)
apply (subgoal_tac "is_class G D")
apply (simp add: comp_wf_mrT)
apply (simp add: wf_prog_def ws_prog_def ws_cdecl_def)
apply blast
```

```
apply (simp add: wf_cdecl_mdecl_def)
apply (simp add: split_beta)
apply (intro strip)


apply (subgoal_tac "∃ sig rT mb. x = (sig, rT, mb)")
apply (erule exE)+
apply (simp (no_asm_simp) add: compMethod_def split_beta)
apply (erule conjE)+
apply (drule_tac x="(sig, rT, mb)" in bspec) apply simp
apply (rule_tac mn="fst sig" and pTs="snd sig" in wt_method_comp)
  apply assumption+
  apply simp
apply (simp (no_asm_simp) add: compTp_def)
apply (simp (no_asm_simp) add: compMethod_def split_beta)
apply (frule WellForm.methd) apply assumption+
apply simp
apply simp
apply (simp add: compMethod_def split_beta)
apply auto
done




declare split_paired_All [simp add]
declare split_paired_Ex [simp add]


end

theory MicroJava
imports
  "J/JTypeSafe"
  "J/Example"
  "J/JListExample"
  "JVM/JVMListExample"
  "JVM/JVMDefensive"
  "BV/LBVJVM"
  "BV/BVNoTypeError"
  "BV/BVExample"
  "Comp/CorrComp"
  "Comp/CorrCompTp"
begin

end
```

Bibliography

[1] G. Klein and T. Nipkow. Verified lightweight bytecode verification. In S. Drossopoulou, S. Eisenbach, B. Jacobs, G. T. Leavens, P. Müller, and A. Poetzsch-Heffter, editors, *Formal Techniques for Java Programs*. Technical Report 269, 5/2000, Fernuniversität Hagen, 2000. ECOOP2000 Workshop proceedings available from http://www.informatik.fernuni-hagen.de/pi5/publications.html.

[2] G. Klein and T. Nipow. Verified lightweight bytecode verification. *Concurrency and Computation: Practice and Experience*, 13(13):1133–1151, 2001. Invited contribution to special issue on Formal Techniques for Java.

[3] T. Nipkow. Verified bytecode verifiers. In F. Honsell, editor, *Foundations of Software Science and Computation Structures (FOSSACS 2001)*, volume 2030, pages 347–363, 2001.

[4] T. Nipkow, D. v. Oheimb, and C. Pusch. μJava: Embedding a programming language in a theorem prover. In F. L. Bauer and R. Steinbrüggen, editors, *Foundations of Secure Computation*, volume 175 of *NATO Science Series F: Computer and Systems Sciences*, pages 117–144. IOS Press, 2000.

[5] D. von Oheimb. Axiomatic semantics for Java$^{\ell ight}$ in Isabelle/HOL. In S. Drossopoulou, S. Eisenbach, B. Jacobs, G. T. Leavens, P. Müller, and A. Poetzsch-Heffter, editors, *Formal Techniques for Java Programs*. Technical Report 269, 5/2000, Fernuniversität Hagen, 2000. ECOOP2000 Workshop proceedings available from http://www.informatik.fernuni-hagen.de/pi5/publications.html.