Java Source and Bytecode Formalizations in Isabelle: μJava

– VerifiCard Project Deliverables –

Gerwin Klein Tobias Nipkow David von Oheimb Leonor Prensa Nieto
Norbert Schirmer Martin Strecker

January 25, 2002

Contents

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 1) 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 2) 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 3) 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 33). 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 35).
- 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 37 to 40).
- For *lightweight bytecode verification*, type checking of bytecode can be reduced to method types with small size.

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.

Chapter 1

Java Source Language

1 Some Auxiliary Definitions

theory *JBasis = Main:*

lemmas *[simp] = Let_def*

1.1 unique

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

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 Java types

theory *Type = JBasis:*

typedecl *cnam*

— exceptions
datatype
  xcpt
  = NullPointer
  | ClassCast
  | OutOfMemory

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

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

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

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

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

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

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

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

end

3 Class Declarations and Programs

theory *Decl = Type:*

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

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

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

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

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

  'c prog = "'c cdecl list" — program

constdefs
  ObjectC :: "'c cdecl" — decl of root class
  "ObjectC ≡ (Object, (arbitrary, [], []))"

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

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

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

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

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

end

4 Relations between Java Types

theory *TypeRel = Decl:*

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

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

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

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

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

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

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

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

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

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

apply *auto*
done

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

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

declare *class_rec.simps [simp del]*

lemma *class_rec_lemma: "⟦ wf ((subcls1 G)^-1); class G C = Some (D,fs,ms)⟧ ⟹*
class_rec (G,C) t f = f C fs ms (if C=Object then t else class_rec (G,D) t f)"
apply *(rule class_rec.simps [THEN trans [THEN fun_cong [THEN fun_cong]]])*
apply *simp*
done

consts

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

— methods of a class, with inheritance, overriding and hiding, cf. 8.4.6
defs *method_def: "method ≡ λ(G,C). class_rec (G,C) empty (λC fs ms ts.*
ts ++ map_of (map (λ(s,m). (s,(C,m))) ms))"

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

— list of fields of a class, including inherited and hidden ones
defs *fields_def: "fields ≡ λ(G,C). class_rec (G,C) [] (λC fs ms ts.*
map (λ(fn,ft). ((fn,C),ft)) fs @ ts)"

lemma *fields_rec_lemma: "[|class G C = Some (D,fs,ms); wf ((subcls1 G)^-1)|] ==>*
fields (G,C) =
map (λ(fn,ft). ((fn,C),ft)) fs @ (if C = Object then [] else fields (G,D))"
apply *(unfold fields_def)*
apply *(simp split del: split_if)*
apply *(erule (1) class_rec_lemma [THEN trans])*
apply *auto*
done

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

lemma *field_fields:*
"field (G,C) fn = Some (fd, fT) ⟹ map_of (fields (G,C)) (fn, fd) = Some fT"

```
apply (unfold field_def)
apply (rule table_of_remap_SomeD)
apply simp
done
```

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

```
inductive "widen G" intros
  refl   [intro!, simp]:       "G⊢      T ≼ T"   — identity conv., cf. 5.1.1
  subcls         : "G⊢C≼C D ==> G⊢Class C ≼ Class D"
  null   [intro!]:             "G⊢     NT ≼ RefT R"
```

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

```
inductive "cast G" intros
  widen:  "G⊢C≼C D ==> G⊢C ≼? D"
  subcls: "G⊢D≼C C ==> G⊢C ≼? D"

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

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

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

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

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

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

theorem widen_trans[trans]: "⟦G⊢S≼U; G⊢U≼T⟧ ⟹ G⊢S≼T"
proof -
  assume "G⊢S≼U" thus "⋀T. G⊢U≼T ⟹ G⊢S≼T"
  proof induct
    case (refl T T') thus "G⊢T≼T'" .
  next
    case (subcls C D T)
    then obtain E where "T = Class E" by (blast dest: widen_Class)
```

```
    with subcls show "G⊢Class C≼T" by (auto elim: rtrancl_trans)
  next
    case (null R RT)
    then obtain rt where "RT = RefT rt" by (blast dest: widen_RefT)
    thus "G⊢NT≼RT" by auto
  qed
qed

end
```

5 Java Values

theory *Value = Type:*

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

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

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

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

primrec
"the_Bool (Bool b) = b"

primrec
"the_Intg (Intg i) = i"

primrec
"the_Addr (Addr a) = a"

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

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

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

end

6 Program State

theory *State = TypeRel + Value:*

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

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

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

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

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

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

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

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

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

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

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

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

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

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

lemma *new_AddrD:*
"(a,x) = new_Addr h ==> h a = None ∧ x = None | x = Some OutOfMemory"
apply *(unfold new_Addr_def)*
apply(*simp add: Pair_fst_snd_eq Eps_split)*
apply(*rule someI)*

```
apply(rule disjI2)
apply(rule_tac "r" = "snd (?a,Some OutOfMemory)" in trans)
apply auto
done


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

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

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

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

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

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

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

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

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

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

done

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

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

end

7 Expressions and Statements

theory *Term = Value:*

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

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

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

end

8 Well-formedness of Java programs

theory *WellForm = TypeRel:*

for static checks on expressions and statements, see WellType.

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

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

simplifications:

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

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

constdefs

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

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

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

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

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

lemma *class_wf:*

```
 "[|class G C = Some c; wf_prog wf_mb G|] ==> wf_cdecl wf_mb G (C,c)"
apply (unfold wf_prog_def class_def)
apply (simp)
apply (fast dest: map_of_SomeD)
done
```

lemma *class_Object [simp]:*

```
  "wf_prog wf_mb G ==> class G Object = Some (arbitrary, [], [])"
apply (unfold wf_prog_def ObjectC_def class_def)
apply (auto intro: map_of_SomeI)
done
```

lemma *is_class_Object [simp]: "wf_prog wf_mb G ==> is_class G Object"*

```
apply (unfold is_class_def)
apply (simp (no_asm_simp))
done
```

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

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

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

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

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

lemma wf_subcls1: "wf_prog wf_mb G ==> wf ((subcls1 G)^-1)"
apply (rule finite_acyclic_wf)
apply ( subst finite_converse)
apply ( rule finite_subcls1)
apply (subst acyclic_converse)
apply (erule acyclic_subcls1)
done

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

lemma subcls1_induct:
"[|is_class G C; wf_prog wf_mb G; P Object;
   !!C D fs ms. [|C ≠ Object; is_class G C; class G C = Some (D,fs,ms) ∧
    wf_cdecl wf_mb G (C,D,fs,ms) ∧ G⊢C≺C1D ∧ is_class G D ∧ P D|] ==> P C
```

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

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

lemmas method_rec = wf_subcls1 [THEN [2] method_rec_lemma]

lemmas fields_rec = wf_subcls1 [THEN [2] fields_rec_lemma]

lemma method_Object [simp]: "wf_prog wf_mb G ==> method (G,Object) = empty"
apply(subst method_rec)
apply auto
done

lemma fields_Object [simp]: "wf_prog wf_mb G ==> fields (G,Object) = []"
apply(subst fields_rec)
apply auto
done

lemma field_Object [simp]: "wf_prog wf_mb G ==> field (G,Object) = empty"
apply (unfold field_def)
apply(simp (no_asm_simp))
done

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

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

done

lemma *widen_fields_defpl': "[|is_class G C; wf_prog wf_mb G|] ==>*
* ∀((fn,fd),fT)∈set (fields (G,C)). G⊢C≼C fd"*
apply(*erule subcls1_induct*)
apply(*assumption*)
apply(*simp (no_asm_simp)*)
apply(*tactic "safe_tac HOL_cs"*)
apply(*subst fields_rec*)
apply(*assumption*)
apply(*assumption*)
apply(*simp (no_asm) split del: split_if*)
apply(*rule ballI*)
apply(*simp (no_asm_simp) only: split_tupled_all*)
apply(*simp (no_asm)*)
apply(*erule UnE*)
apply(*force*)
apply(*erule r_into_rtrancl [THEN rtrancl_trans]*)
apply *auto*
done

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

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

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

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

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

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

lemma *subcls_widen_methd [rule_format (no_asm)]:*
* "[|G⊢T≼C T'; wf_prog wf_mb G|] ==>*
*   ∀D rT b. method (G,T') sig = Some (D,rT ,b) -->*
* (∃D' rT' b'. method (G,T) sig = Some (D',rT',b') ∧ G⊢rT'≼rT)"*
apply(*drule rtranclD)*
apply(*erule disjE)*
apply(*fast)*
apply(*erule conjE)*
apply(*erule trancl_trans_induct)*
prefer *2*
apply(*clarify)*
apply(*drule spec, drule spec, drule spec, erule (1) impE)*
apply(*fast elim: widen_trans)*
apply(*clarify)*

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

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

lemma method_in_md [rule_format (no_asm)]:
  "wf_prog wf_mb G ==> is_class G C ⟹ ∀D. method (G,C) sig = Some(D,mh,code)
  --> is_class G D ∧ method (G,D) sig = Some(D,mh,code)"
apply (erule (1) subcls1_induct)
 apply (simp)
apply (erule conjE)
apply (subst method_rec)
  apply (assumption)
 apply (assumption)
apply (clarify)
apply (erule_tac "x" = "Da" in allE)
apply (clarsimp)
 apply (simp add: map_of_map)
 apply (clarify)
 apply (subst method_rec)
   apply (assumption)
  apply (assumption)
 apply (simp add: override_def map_of_map split add: option.split)
done

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

lemma Call_lemma:
"[|method (G,C) sig = Some (md,rT,b); G⊢T''≼C C; wf_prog wf_mb G;
  class G C = Some y|] ==> ∃T' rT' b. method (G,T'') sig = Some (T',rT',b) ∧
  G⊢rT'≼rT ∧ G⊢T''≼C T' ∧ wf_mhead G sig rT' ∧ wf_mb G T' (sig,rT',b)"
```

apply(*drule (2) widen_methd)*
apply(*clarify)*
apply(*frule subcls_is_class2)*
apply *(unfold is_class_def)*
apply *(simp (no_asm_simp))*
apply(*drule method_wf_mdecl)*
apply(*unfold wf_mdecl_def)*
apply(*unfold is_class_def)*
apply *auto*
done

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

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

lemma *methd:*
"[| wf_prog wf_mb G; (C,S,fs,mdecls) ∈ set G; (sig,rT,code) ∈ set mdecls |]
==> method (G,C) sig = Some(C,rT,code) ∧ is_class G C"
proof -
assume *wf: "wf_prog wf_mb G"*
assume *C: "(C,S,fs,mdecls) ∈ set G"*

assume *m: "(sig,rT,code) ∈ set mdecls"*
moreover
from *wf* **have** *"class G Object = Some (arbitrary, [], [])"* **by** *simp*
moreover
from *wf C* **have** *c: "class G C = Some (S,fs,mdecls)"*
by *(auto simp add: wf_prog_def class_def is_class_def intro: map_of_SomeI)*
ultimately
have *0: "C ≠ Object"* **by** *auto*

from *wf C* **have** *"unique mdecls"* **by** *(unfold wf_prog_def wf_cdecl_def) auto*
hence *"unique (map (λ(s,m). (s,C,m)) mdecls)"* **by** *(induct mdecls, auto)*
with *m* **have** *"map_of (map (λ(s,m). (s,C,m)) mdecls) sig = Some (C,rT,code)"*

```
    by (force intro: map_of_SomeI)
  with wf C m c D
  show ?thesis by (auto simp add: is_class_def dest: method_rec)
qed

end
```

9 Well-typedness Constraints

theory *WellType = Term + WellForm:*

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

simplifications:

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

local variables, including method parameters and This:

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

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

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

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

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

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

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

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

lemma *appl_methsD:*
```
"((md,rT),pTs')∈appl_methds G C (mn, pTs) ==>
  ∃D b. md = Class D ∧ method (G,C) (mn, pTs') = Some (D,rT,b)
  ∧ list_all2 (λT T'. G⊢T⪯T') pTs pTs'"
```

apply *(unfold appl_methds_def)*
apply *(fast)*
done

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

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

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

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

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

types
java_mb = "vname list × (vname × ty) list × stmt × expr"
— method body with parameter names, local variables, block, result expression.
— local variables might include This, which is hidden anyway

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

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

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

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

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

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

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

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

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


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

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

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

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

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


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

— well-typed expression lists

  — cf. 15.11.???
  Nil: "E⊢[][::][]"

  — cf. 15.11.???
  Cons:"[| E⊢e::T;
           E⊢es[::]Ts |] ==>
        E⊢e#es[::]T#Ts"

— well-typed statements
```

```
  Skip:"E⊢Skip√"

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

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

— cf. 14.8

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

— cf. 14.10

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

constdefs

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

syntax

```
 wf_java_prog :: "java_mb prog => bool"
```

translations

```
  "wf_java_prog" == "wf_prog wf_java_mdecl"
```

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

lemmas `ty_expr_is_type = wt_is_type [THEN conjunct1,THEN mp, COMP swap_prems_rl]`

end

10 Operational Evaluation (big step) Semantics

theory *Eval = State + WellType:*

consts

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

syntax *(xsymbols)*

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

syntax

```
  eval :: "[java_mb prog,xstate,expr,val,xstate] => bool "
          ("_ |- _ -_>_-> _" [51,82,60,82,82] 81)
  evals:: "[java_mb prog,xstate,expr list,
                        val list,xstate] => bool "
          ("_ |- _ -_[>]_-> _" [51,82,60,51,82] 81)
  exec :: "[java_mb prog,xstate,stmt,    xstate] => bool "
          ("_ |- _ -_-> _" [51,82,60,82] 81)
```

translations

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

inductive *"eval G" "evals G" "exec G"* **intros**

```
  XcptE:"G⊢(Some xc,s) -e≻arbitrary-> (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[≻]arbitrary-> (Some xc,s)"
```

— cf. 15.11.???

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

— cf. 15.6.4

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

— execution of statements

— cf. 14.1

```
XcptS:"G⊢(Some xc,s) -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 *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

11 Conformity Relations for Type Soundness Proof

theory *Conform = State + WellType:*

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

constdefs

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

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

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

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

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

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


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

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

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

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

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

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

11.1 hext

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

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

apply *(force)*
done

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

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

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

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

11.2 conf

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

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

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

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

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

lemma *conf_upd_obj:*
"h a = Some (C,fs) ==> (G,h(a↦(C,fs'))⊢x::≤T) = (G,h⊢x::≤T)"
apply *(unfold conf_def)*

```
apply (rule val.induct)
apply auto
done

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

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

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

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

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

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

lemma non_np_objD: "!!G. [|a' ≠ Null; G,h⊢a'::≼ Class C; C ≠ Object|] ==>
  (∃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 -->
  (∀C. t = ClassT C --> C ≠ Object) -->
  (∃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

11.3 lconf

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

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

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

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

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

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

lemma *lconf_ext_list [rule_format (no_asm)]:*
"G,h⊢l[::≼]L ==> ∀vs Ts. nodups 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

11.4 oconf

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

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

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

11.5 hconf

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

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

11.6 conforms

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

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

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

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

lemma *conforms_upd_obj:*
"[|(h,l)::≼(G, lT); G,h(a↦obj)⊢obj√; h≤|h(a↦obj)|] ==> (h(a↦obj),l)::≼(G, lT)"
apply(*rule conforms_hext*)
apply *auto*
apply(*rule hconfI*)
apply(*drule conforms_heapD*)

apply(*tactic {* auto_tac (HOL_cs addEs [thm "oconf_hext"]*
*addDs [thm "hconfD"], simpset() delsimps [split_paired_All]) *}*)
done

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

end

12 Type Safety Proof

theory *JTypeSafe = Eval + Conform:*

declare *split_beta [simp]*

lemma *NewC_conforms:*
"[|h a = None; (h, l)::≼(G, lT); wf_prog wf_mb G; is_class G C|] ==>
(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)*
done

lemma *Cast_conf:*
"[| wf_prog wf_mb G; G,h⊢v::≼Class C; G⊢C≼? D; cast_ok G D h v|]
==> G,h⊢v::≼Class D"
apply *(unfold cast_ok_def)*
apply(*case_tac "v = Null")*
apply(*simp)*
apply(*drule widen_RefT)*
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); (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(*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;
(h, l)::≼(G, lT); G,h⊢x::≼T'; np a' x' = None|] ==>
h''≤|h(a↦(c,(fs((fn,fd)↦x)))) ∧
(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(*force)*
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( 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; nodups 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; (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. (h, init_vars lvars(pns[↦]pvs)(This↦a'))::⪯(G, lT) -->
  (G, lT)⊢blk√ -->  h≤|xi ∧  (xi, xl)::⪯(G, lT);
  ∀lT. (xi, xl)::⪯(G, lT) --> (∀T. (G, lT)⊢res::T -->
         xi≤|h' ∧ (h', xj)::⪯(G, lT) ∧ (x' = None --> G,h'⊢v::⪯T));
  G,xh⊢a'::⪯ Class C |] ==>
  xc≤|h' ∧ (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( 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)
apply(  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( 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( erule conforms_hext)
apply(  erule (1) hext_trans)
apply( erule conforms_heapD)
done

declare split_if [split del]
declare fun_upd_apply [simp del]
declare fun_upd_same [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)]))
*}
ML{*
Unify.search_bound := 40;
Unify.trace_bound  := 40
*}
theorem eval_evals_exec_type_sound:
"wf_java_prog G ==>
  (G⊢(x,(h,l)) -e ≻v -> (x', (h',l')) -->
      (∀lT.   (h ,l )::≼(G,lT) --> (∀T . (G,lT)⊢e  :: T -->
      h≤|h' ∧ (h',l')::≼(G,lT) ∧ (x'=None --> G,h'⊢v  ::≼ T )))) ∧
  (G⊢(x,(h,l)) -es[≻]vs-> (x', (h',l')) -->
      (∀lT.   (h ,l )::≼(G,lT) --> (∀Ts. (G,lT)⊢es[::]Ts -->
      h≤|h' ∧ (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.   (h ,l )::≼(G,lT) -->       (G,lT)⊢c  √ -->
      h≤|h' ∧ (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 (thms "ty_expr_ty_exprs_wt_stmt.elims")*
*THEN_ALL_NEW Full_simp_tac) *}*)
apply(*tactic "ALLGOALS (EVERY' [REPEAT o (etac conjE), REPEAT o hyp_subst_tac])"*)

— Level 7

— 15 NewC
apply(*drule new_AddrD*)
apply(*erule disjE*)
prefer *2*
apply(*simp (no_asm_simp)*)
apply(*clarsimp*)
apply(*rule conjI*)
apply(*force elim!: NewC_conforms*)
apply(*rule conf_obj_AddrI*)
apply(*rule_tac [2] 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(*fast elim: conforms_localD [THEN lconfD]*)

— for FAss
apply(*tactic {* EVERY'[eresolve_tac (thms "ty_expr_ty_exprs_wt_stmt.elims")*
THEN_ALL_NEW Full_simp_tac, REPEAT o (etac conjE), hyp_subst_tac] 3}*)

— for if
apply(*tactic {* (case_tac "the_Bool v" THEN_ALL_NEW Asm_full_simp_tac) 8*}*)

apply (*tactic "forward_hyp_tac"*)

— 11+1 if
prefer *8*
apply(*fast intro: hext_trans*)
prefer *8*
apply(*fast intro: hext_trans*)

— 10 Expr
prefer *6*
apply(*fast*)

— 9 ???
apply(*simp_all*)

— 8 Cast
prefer *8*
apply (*rule impI*)
apply (*drule raise_if_NoneD*)
apply (*clarsimp*)
apply (*fast elim: Cast_conf*)

— 7 LAss
apply (*fold fun_upd_def*)
apply(*tactic {* (eresolve_tac (thms "ty_expr_ty_exprs_wt_stmt.elims")*
*THEN_ALL_NEW Full_simp_tac) 1 *}*)
apply(*blast intro: conforms_upd_local conf_widen*)

— 6 FAcc
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 Cons
prefer *3*
apply(*fast dest: evals_no_xcpt intro: conf_hext hext_trans*)

— 3 ;;
prefer *3*
apply(*fast intro: hext_trans*)

— 2 FAss
apply(*case_tac "x2 = None"*)
prefer *2*
apply(*simp (no_asm_simp)*)
apply(*fast intro: hext_trans*)
apply(*simp*)
apply(*drule eval_no_xcpt*)
apply(*erule FAss_type_sound, rule HOL.refl, assumption+*)

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(  fast elim: hext_trans)
apply( drule (1) ty_expr_is_type)
apply(clarsimp)
apply(unfold is_class_def)
apply(clarsimp)
apply(rule Call_type_sound)
prefer 11
apply blast
apply (simp (no_asm_simp))+
done
ML{*
Unify.search_bound := 20;
Unify.trace_bound  := 20
*}

lemma eval_type_sound: "!!E s s'.
  [| G=prg E; wf_java_prog G; G⊢(x,s) -e≻v -> (x',s'); s::≼E; E⊢e::T |]
  ==> s'::≼E ∧ (x'=None --> G,heap s'⊢v::≼T)"
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 exec_type_sound: "!!E s s'.
  [| G=prg E; wf_java_prog G; G⊢(x,s) -s0-> (x',s'); s::≼E; E⊢s0√ |]
  ==> s'::≼E"
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;
          s::≼E; E⊢e::Class C; method (G,C) sig ≠ None|] ==>
  method (G,fst (the (heap s' (the_Addr a')))) sig ≠ None"
apply( drule (4) eval_type_sound)
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]

end
```

13 Example MicroJava Program

theory *Example = Eval:*

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]*

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

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

axioms
```
  Base_not_Object: "Base ≠ Object"
  Ext_not_Object:  "Ext  ≠ Object"
  e_not_This:      "e ≠ This"
```

declare *Base_not_Object [simp] Ext_not_Object [simp]*
declare *e_not_This [simp]*
declare *Base_not_Object [THEN not_sym, simp]*
declare *Ext_not_Object [THEN not_sym, 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]))"
```

syntax

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

translations
```
  "NP"   == "NullPointer"
  "tprg" == "[ObjectC, BaseC, ExtC]"
  "obj1"    <= "(Ext, empty((vee, Base)↦Bool False)
         ((vee, Ext )↦Intg 0))"
  "s0" == " Norm    (empty, empty)"
  "s1" == " Norm    (empty(a↦obj1),empty(e↦Addr a))"
  "s2" == " Norm    (empty(a↦obj1),empty(x↦Null)(This↦Addr a))"
  "s3" == "(Some NP, empty(a↦obj1),empty(e↦Addr a))"
```

ML *{* bind_thm ("map_of_Cons", hd (tl (thms "map_of.simps"))) *}*
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 (arbitrary, [], [])"*
apply *(unfold ObjectC_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 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"
apply *(unfold ObjectC_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)
done

lemmas subcls_direct = subcls1I [THEN r_into_rtrancl]

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:
"wf_cdecl wf_java_mdecl tprg ObjectC"
apply (unfold wf_cdecl_def wf_fdecl_def ObjectC_def)
apply (simp (no_asm))
done

lemma wf_BaseC:
"wf_cdecl wf_java_mdecl tprg BaseC"
apply (unfold wf_cdecl_def wf_fdecl_def BaseC_def)
apply (simp (no_asm))
apply (fold BaseC_def)
apply (rule wf_foo_Base [THEN conjI])
apply auto
done

lemma wf_ExtC:
"wf_cdecl wf_java_mdecl tprg ExtC"
apply (unfold wf_cdecl_def wf_fdecl_def ExtC_def)
apply (simp (no_asm))
apply (fold ExtC_def)
apply (rule wf_foo_Ext [THEN conjI])
apply auto
apply (drule rtranclD)
apply auto
done

lemma wf_tprg:
"wf_prog wf_java_mdecl tprg"
apply (unfold wf_prog_def Let_def)
apply(simp add: wf_ObjectC wf_BaseC wf_ExtC unique_classes)
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))
apply (subgoal_tac "tprg⊢NT⪯ Class Base")
apply  (auto simp add: map_of_Cons foo_Base_def)
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 {* fun t thm = resolve_tac (thms "ty_expr_ty_exprs_wt_stmt.intros") 1 thm *}
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 {* fun e t = resolve_tac (thm "NewCI"::thms "eval_evals_exec.intros") 1 t *}

declare split_if [split del]
declare init_vars_def [simp] c_hupd_def [simp] cast_ok_def [simp]
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
```

14 Example for generating executable code from Java semantics

theory *JListExample = Eval:*

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
```

constdefs

```
  val_name :: vname
  "val_name == VName val_nam"

  next_name :: vname
  "next_name == VName next_nam"

  l_name :: vname
  "l_name == VName l_nam"

  l1_name :: vname
  "l1_name == VName l1_nam"

  l2_name :: vname
  "l2_name == VName l2_nam"

  l3_name :: vname
  "l3_name == VName l3_nam"

  l4_name :: vname
  "l4_name == VName l4_nam"

  list_class :: "java_mb class"
  "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))])"

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

types_code

```
  cname ("string")
  vnam ("string")
  mname ("string")
```

```
  loc ("int")
```

consts_code

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

  "wf" ("true?")

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

ML *{**

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

generate_code

```
  test = "example_prg⊢Norm (Map.empty, Map.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])))-> _"
```

15 Big step execution

ML *{**

```
val Library.Some ((_, (heap, locs)), _) = Seq.pull test;
locs l1_name;
locs l2_name;
locs l3_name;
locs l4_name;
snd (the (heap 0)) (val_name, "list");
snd (the (heap 0)) (next_name, "list");
snd (the (heap 1)) (val_name, "list");
snd (the (heap 1)) (next_name, "list");
snd (the (heap 2)) (val_name, "list");
```

```
snd (the (heap 2)) (next_name, "list");
snd (the (heap 3)) (val_name, "list");
snd (the (heap 3)) (next_name, "list");

*}
```

end

Chapter 2

Java Virtual Machine

16 State of the JVM

theory *JVMState = Conform:*

16.1 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

16.2 Exceptions

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

— redefines State.new_Addr:
```
  new_Addr :: "aheap => loc × val option"
  "new_Addr h == SOME (a,x). (h a = None ∧  x = None) |
                             x = raise_system_xcpt True OutOfMemory"
```

16.3 Runtime State

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

16.4 Lemmas

lemma *new_AddrD:*
```
  "new_Addr hp = (ref, xcp) ⟹ hp ref = None ∧ xcp = None ∨ xcp = Some (Addr (XcptRef OutOfMe
  apply (drule sym)
  apply (unfold new_Addr_def)
  apply (simp add: raise_system_xcpt_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 *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
```

17 Instructions of the JVM

theory *JVMInstructions = JVMState:*

datatype

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

types

```
  bytecode = "instr list"
  exception_entry = "p_count × p_count × p_count × cname"
                  — start-pc, end-pc, handler-pc, exception type
  exception_table = "exception_entry list"
  jvm_method = "nat × nat × bytecode × exception_table"
  jvm_prog = "jvm_method prog"
```

end

18 JVM Instruction Semantics

theory *JVMExecInstr = JVMInstructions + JVMState:*

consts

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

primrec

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

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

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

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

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

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

 "exec_instr (Invoke C mn ps) G hp stk vars Cl sig pc frs =
  (let n    = length ps;
       argsoref = take (n+1) stk;
       oref = last argsoref;
       xp'  = raise_system_xcpt (oref=Null) NullPointer;
```

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

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

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

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

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

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

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

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

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

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

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

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

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

```
   in
      (xcpt', hp, (stk, vars, Cl, sig, pc)#frs))"
```

end

19 Exception handling in the JVM

theory *JVMExceptions = JVMInstructions:*

constdefs

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

consts

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

primrec

```
"match_exception_table G cn pc []     = None"
"match_exception_table G cn pc (e#es) = (if match_exception_entry G cn pc e
                                          then Some (fst (snd (snd e)))
                                          else match_exception_table G cn pc es)"
```

consts

```
cname_of :: "aheap ⇒ val ⇒ cname"
ex_table_of :: "jvm_method ⇒ exception_table"
```

translations

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

consts

```
find_handler :: "jvm_prog ⇒ val option ⇒ aheap ⇒ 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, and they don't carry any information other than their type:

constdefs

```
preallocated :: "aheap ⇒ bool"
"preallocated hp ≡ ∀x. hp (XcptRef x) = Some (Xcpt x, empty)"
```

lemma *preallocated_iff [iff]:*

```
"preallocated hp ⟹ hp (XcptRef x) = Some (Xcpt x, empty)"
```

by *(unfold preallocated_def) fast*

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

end

20 Program Execution in the JVM

theory *JVMExec = JVMExecInstr + JVMExceptions:*

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

— exec is not recursive. recdef is just used for pattern matching
recdef *exec "{}"*
```
  "exec (G, xp, hp, []) = None"

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

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

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

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

end

21 Example for generating executable code from JVM semantics

theory *JVMListExample = JVMExec:*

consts

```
  list_name :: cname
  test_name :: cname
  append_name :: mname
  makelist_name :: mname
  val_nam :: vnam
  next_nam :: vnam
```

constdefs

```
  val_name :: vname
  "val_name == VName val_nam"

  next_name :: vname
  "next_name == VName next_nam"

  list_class :: "jvm_method class"
  "list_class ==
    (Object,
     [(val_name, PrimT Integer), (next_name, RefT (ClassT list_name))],
     [((append_name, [RefT (ClassT list_name)]), PrimT Integer,
      (0, 0,
       [Load 0,
        Getfield next_name list_name,
        Dup,
        LitPush Null,
        Ifcmpeq 4,
        Load 1,
        Invoke list_name append_name [RefT (ClassT list_name)],
        Return,
        Pop,
        Load 0,
        Load 1,
        Putfield next_name list_name,
        LitPush Unit,
        Return],[]))])"

  test_class :: "jvm_method class"
  "test_class ==
    (Object, [],
     [((makelist_name, []), PrimT Integer,
      (0, 3,
       [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 [RefT (ClassT list_name)],
        Load 0,
        Load 2,
        Invoke list_name append_name [RefT (ClassT list_name)],
        LitPush Unit,
        Return],[]))])"

  example_prg :: jvm_prog
  "example_prg == [ObjectC, (list_name, list_class), (test_name, test_class)]"

  start_state :: jvm_state
  "start_state ==
    (None, empty, [([], Unit # Unit # Unit # [], test_name, (makelist_name, []), 0)])"
```

types_code

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

consts_code

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

  "wf" ("true?")

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

  "Object" (""Object"")
  "list_name" (""list"")
  "test_name" (""test"")
  "append_name" (""append"")
  "makelist_name" (""makelist"")
  "val_nam" (""val"")
  "next_nam" (""next"")
```

ML *{*

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

}

22 Single step execution

generate_code

```
  test = "exec (example_prg, start_state)"
```

ML *{* test *}*
ML *{* exec (example_prg, the it) *}*
ML *{* exec (example_prg, the it) *}*
ML *{* exec (example_prg, the it) *}*
ML *{* exec (example_prg, the it) *}*
ML *{* exec (example_prg, the it) *}*
ML *{* exec (example_prg, the it) *}*

```
ML {* exec (example_prg, the it) *}
ML {* exec (example_prg, the it) *}
ML {* exec (example_prg, the it) *}
ML {* exec (example_prg, the it) *}
ML {* exec (example_prg, the it) *}
ML {* exec (example_prg, the it) *}
ML {* exec (example_prg, the it) *}
ML {* exec (example_prg, the it) *}
ML {* exec (example_prg, the it) *}
ML {* exec (example_prg, the it) *}
ML {* exec (example_prg, the it) *}
ML {* exec (example_prg, the it) *}
ML {* exec (example_prg, the it) *}
ML {* exec (example_prg, the it) *}
ML {* exec (example_prg, the it) *}
ML {* exec (example_prg, the it) *}
ML {* exec (example_prg, the it) *}
ML {* exec (example_prg, the it) *}
ML {* exec (example_prg, the it) *}
ML {* exec (example_prg, the it) *}
ML {* exec (example_prg, the it) *}
ML {* exec (example_prg, the it) *}
ML {* exec (example_prg, the it) *}
ML {* exec (example_prg, the it) *}
ML {* exec (example_prg, the it) *}
ML {* exec (example_prg, the it) *}
ML {* exec (example_prg, the it) *}
ML {* exec (example_prg, the it) *}
ML {* exec (example_prg, the it) *}
ML {* exec (example_prg, the it) *}
ML {* exec (example_prg, the it) *}
ML {* exec (example_prg, the it) *}
ML {* exec (example_prg, the it) *}
ML {* exec (example_prg, the it) *}
ML {* exec (example_prg, the it) *}
ML {* exec (example_prg, the it) *}
ML {* exec (example_prg, the it) *}
ML {* exec (example_prg, the it) *}
ML {* exec (example_prg, the it) *}
ML {* exec (example_prg, the it) *}
ML {* exec (example_prg, the it) *}
ML {* exec (example_prg, the it) *}
ML {* exec (example_prg, the it) *}
ML {* exec (example_prg, the it) *}
ML {* exec (example_prg, the it) *}
ML {* exec (example_prg, the it) *}
ML {* exec (example_prg, the it) *}

end
```

Chapter 3

Bytecode Verifier

23 A general "while" combinator Tobias Nipkow

theory *While_Combinator = Main:*

We define a while-combinator *while* and prove: (a) an unrestricted unfolding law (even if while diverges!) (I got this idea from Wolfgang Goerigk), and (b) the invariant rule for reasoning about *while*.

```
consts while_aux :: "('a => bool) × ('a => 'a) × 'a => 'a"
recdef (permissive) while_aux
  "same_fst (λb. True) (λb. same_fst (λc. True) (λc.
      {(t, s).  b s ∧ c s = t ∧
        ¬ (∃f. f (0::nat) = s ∧ (∀i. b (f i) ∧ c (f i) = f (i + 1)))}))"
  "while_aux (b, c, s) =
    (if (∃f. f (0::nat) = s ∧ (∀i. b (f i) ∧ c (f i) = f (i + 1)))
      then arbitrary
      else if b s then while_aux (b, c, c s)
      else s)"

recdef_tc while_aux_tc: while_aux
  apply (rule wf_same_fst)
  apply (rule wf_same_fst)
  apply (simp add: wf_iff_no_infinite_down_chain)
  apply blast
  done

constdefs
  while :: "('a => bool) => ('a => 'a) => 'a => 'a"
  "while b c s == while_aux (b, c, s)"

lemma while_aux_unfold:
  "while_aux (b, c, s) =
    (if ∃f. f (0::nat) = s ∧ (∀i. b (f i) ∧ c (f i) = f (i + 1))
      then arbitrary
      else if b s then while_aux (b, c, c s)
      else s)"
  apply (rule while_aux_tc [THEN while_aux.simps [THEN trans]])
  apply (rule refl)
  done
```

The recursion equation for *while*: directly executable!

```
theorem while_unfold [code]:
    "while b c s = (if b s then while b c (c s) else s)"
  apply (unfold while_def)
  apply (rule while_aux_unfold [THEN trans])
  apply auto
  apply (subst while_aux_unfold)
  apply simp
  apply clarify
  apply (erule_tac x = "λi. f (Suc i)" in allE)
  apply blast
  done

hide const while_aux
```

The proof rule for *while*, where *P* is the invariant.

```
theorem while_rule_lemma[rule_format]:
  "[| !!s. P s ==> b s ==> P (c s);
      !!s. P s ==> ¬ b s ==> Q s;
      wf {(t, s). P s ∧ b s ∧ t = c s} |] ==>
    P s --> Q (while b c s)"
proof -
  case rule_context
  assume wf: "wf {(t, s). P s ∧ b s ∧ t = c s}"
  show ?thesis
    apply (induct s rule: wf [THEN wf_induct])
    apply simp
    apply clarify
    apply (subst while_unfold)
    apply (simp add: rule_context)
    done
qed

theorem while_rule:
  "[| P s;
      !!s. [| P s; b s  |] ==> P (c s);
      !!s. [| P s; ¬ b s  |] ==> Q s;
      wf r;
      !!s. [| P s; b s  |] ==> (c s, s) ∈ r |] ==>
   Q (while b c s)"
apply (rule while_rule_lemma)
prefer 4 apply assumption
apply blast
apply blast
apply(erule wf_subset)
apply blast
done
```

An application: computation of the *lfp* on finite sets via iteration.

```
theorem lfp_conv_while:
  "[| mono f; finite U; f U = U |] ==>
    lfp f = fst (while (λ(A, fA). A ≠ fA) (λ(A, fA). (fA, f fA)) ({}, f {}))"
apply (rule_tac P = "λ(A, B). (A ⊆ U ∧ B = f A ∧ A ⊆ B ∧ B ⊆ lfp f)" and
               r = "((Pow U × UNIV) × (Pow U × UNIV)) ∩
                    inv_image finite_psubset (op - U o fst)" in while_rule)
   apply (subst lfp_unfold)
    apply assumption
   apply (simp add: monoD)
  apply (subst lfp_unfold)
   apply assumption
  apply clarsimp
  apply (blast dest: monoD)
 apply (fastsimp intro!: lfp_lowerbound)
 apply (blast intro: wf_finite_psubset Int_lower2 [THEN [2] wf_subset])
apply (clarsimp simp add: inv_image_def finite_psubset_def order_less_le)
apply (blast intro!: finite_Diff dest: monoD)
done
```

An example of using the *while* combinator.[1]

```
theorem "P (lfp (λN::int set. {0} ∪ {(n + 2) mod 6 | n. n ∈ N})) = P {0, 4, 2}"
proof -
  have aux: "!!f A B. {f n | n. A n ∨ B n} = {f n | n. A n} ∪ {f n | n. B n}"
    apply blast
    done
  show ?thesis
    apply (subst lfp_conv_while [where ?U = "{0, 1, 2, 3, 4, 5}"])
       apply (rule monoI)
      apply blast
     apply simp
    apply (simp add: aux set_eq_subset)
```

The fixpoint computation is performed purely by rewriting:

```
    apply (simp add: while_unfold aux set_eq_subset del: subset_empty)
    done
qed

end
```

[1] It is safe to keep the example here, since there is no effect on the current theory.

24 Semilattices

theory *Semilat = While_Combinator:*

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

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

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

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

order :: "'a ord => bool"
"order r == (!x. x <=_r x) &
(!x y. x <=_r y & y <=_r x --> x=y) &
(!x y z. x <=_r y & y <=_r z --> x <=_r z)"

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

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

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

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

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

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

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

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

```
lemma order_antisym:
  "[| order r; x <=_r y; y <=_r x |] ==> x = y"
apply (unfold order_def)
apply (simp (no_asm_simp))
done

lemma order_trans:
   "[| order r; x <=_r y; y <=_r z |] ==> x <=_r z"
apply (unfold order_def)
apply blast
done

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

lemma order_less_trans:
  "[| order r; x <_r y; y <_r z |] ==> x <_r z"
apply (unfold order_def lesssub_def)
apply blast
done

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

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

lemma semilat_Def:
"semilat(A,r,f) == order r & closed A f &
                 (!x:A. !y:A. x <=_r x +_f y) &
                 (!x:A. !y:A. y <=_r x +_f y) &
                 (!x:A. !y:A. !z:A. x <=_r z & y <=_r z --> x +_f y <=_r z)"
apply (unfold semilat_def split_conv [THEN eq_reflection])
apply (rule refl [THEN eq_reflection])
done

lemma semilatDorderI [simp, intro]:
  "semilat(A,r,f) ==> order r"
  by (simp add: semilat_Def)

lemma semilatDclosedI [simp, intro]:
  "semilat(A,r,f) ==> closed A f"
apply (unfold semilat_Def)
apply simp
done

lemma semilat_ub1 [simp]:
  "[| semilat(A,r,f); x:A; y:A |] ==> x <=_r x +_f y"
  by (unfold semilat_Def, simp)

lemma semilat_ub2 [simp]:
  "[| semilat(A,r,f); x:A; y:A |] ==> y <=_r x +_f y"
  by (unfold semilat_Def, simp)
```

lemma *semilat_lub [simp]:*
"[| x <=_r z; y <=_r z; semilat(A,r,f); x:A; y:A; z:A |] ==> x +_f y <=_r z"
by *(unfold semilat_Def, simp)*

lemma *plus_le_conv [simp]:*
"[| x:A; y:A; z:A; semilat(A,r,f) |]
==> (x +_f y <=_r z) = (x <=_r z & y <=_r z)"
apply *(unfold semilat_Def)*
apply *(blast intro: semilat_ub1 semilat_ub2 semilat_lub order_trans)*
done

lemma *le_iff_plus_unchanged:*
"[| x:A; y:A; semilat(A,r,f) |] ==> (x <=_r y) = (x +_f y = y)"
apply *(rule iffI)*
apply *(intro semilatDorderI order_antisym semilat_lub order_refl semilat_ub2, assumption+)*
apply *(erule subst)*
apply *simp*
done

lemma *le_iff_plus_unchanged2:*
"[| x:A; y:A; semilat(A,r,f) |] ==> (x <=_r y) = (y +_f x = y)"
apply *(rule iffI)*
apply *(intro semilatDorderI order_antisym semilat_lub order_refl semilat_ub1, assumption+)*
apply *(erule subst)*
apply *simp*
done

lemma *closedD:*
"[| closed A f; x:A; y:A |] ==> x +_f y : A"
apply *(unfold closed_def)*
apply *blast*
done

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

lemma *is_lubD:*
"is_lub r x y u ==> is_ub r x y u & (!z. is_ub r x y z --> (u,z):r)"
by *(simp add: is_lub_def)*

lemma *is_ubI:*
"[| (x,u) : r; (y,u) : r |] ==> is_ub r x y u"
by *(simp add: is_ub_def)*

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

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

lemma *is_lub_bigger2 [iff]:*

```
  "is_lub (r^* ) x y x = ((y,x):r^* )"
apply (unfold is_lub_def is_ub_def)
apply blast
done

lemma extend_lub:
  "[| single_valued r; is_lub (r^* ) x y u; (x',x) : r |]
  ==> EX v. is_lub (r^* ) x' y v"
apply (unfold is_lub_def is_ub_def)
apply (case_tac "(y,x) : r^*")
 apply (case_tac "(y,x') : r^*")
  apply blast
 apply (blast elim: converse_rtranclE dest: single_valuedD)
apply (rule exI)
apply (rule conjI)
 apply (blast intro: converse_rtrancl_into_rtrancl dest: single_valuedD)
apply (blast intro: rtrancl_into_rtrancl converse_rtrancl_into_rtrancl
             elim: converse_rtranclE dest: single_valuedD)
done

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

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

lemma is_lub_some_lub:
  "[| single_valued r; acyclic r; (x,u):r^*; (y,u):r^* |]
  ==> is_lub (r^* ) x y (some_lub (r^* ) x y)"
  by (fastsimp dest: single_valued_has_lubs simp add: some_lub_conv)
```

25 An executable lub-finder

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


lemma acyclic_single_valued_finite:
 "⟦acyclic r; single_valued r; (x,y) ∈ r^*⟧
  ⟹ finite (r ∩ {a. (x, a) ∈ r^*} × {b. (b, y) ∈ r^*})"
apply(erule converse_rtrancl_induct)
 apply(rule_tac B = "{}" in finite_subset)
  apply(simp only:acyclic_def)
  apply(blast intro:rtrancl_into_trancl2 rtrancl_trancl_trancl)
 apply simp
```

```
apply(rename_tac x x')
apply(subgoal_tac "r ∩ {a. (x,a) ∈ r*} × {b. (b,y) ∈ r*} =
                   insert (x,x') (r ∩ {a. (x', a) ∈ r*} × {b. (b, y) ∈ r*})")
 apply simp
apply(blast intro:converse_rtrancl_into_rtrancl
            elim:converse_rtranclE dest:single_valuedD)
done


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

lemma is_lub_exec_lub:
  "⟦ single_valued r; acyclic r; (x,u):r^*; (y,u):r^*; !x y. (x,y) ∈ r ⟶ f x = y ⟧
  ⟹ is_lub (r^* ) x y (exec_lub r f x y)"
  by (fastsimp dest: single_valued_has_lubs simp add: exec_lub_conv)

end
```

26 The Error Type

theory *Err = Semilat:*

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

lemma semilat_errI:
  "semilat(A,r,f) ==> semilat(err A, Err.le r, lift2(%x y. OK(f x y)))"
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:
  "!!L. semilat L ==> err_semilat(esl L)"
apply (unfold sl_def esl_def)
apply (simp (no_asm_simp) only: split_tupled_all)
apply (simp add: semilat_errI)
done

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)
```

26.1 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)*

26.2 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

26.3 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: le_iff_plus_unchanged [THEN iffD1] le_iff_plus_unchanged2 [THEN iffD1])*

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

lemma *semilat_le_err_OK1:*
"[| x:A; y:A; semilat(err A, le r, f); OK x +_f OK y = OK z |]
==> x <=_r z"
apply *(rule OK_le_err_OK [THEN iffD1])*
apply *(erule subst)*
apply *simp*
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*
done

```
lemma eq_order_le:
  "[| x=y; order r |] ==> x <=_r y"
apply (unfold order_def)
apply blast
done

lemma OK_plus_OK_eq_Err_conv [simp]:
  "[| x:A; y:A; semilat(err A, le r, fe) |] ==>
  ((OK x) +_fe (OK y) = Err) = (~(? z:A. x <=_r z & y <=_r z))"
proof -
  have plus_le_conv3: "!!A x y z f r.
    [| semilat (A,r,f); x +_f y <=_r z; x:A; y:A; z:A |]
    ==> x <=_r z ∧ y <=_r z"
    by (rule plus_le_conv [THEN iffD1])
  case rule_context
  thus ?thesis
  apply (rule_tac iffI)
   apply clarify
   apply (drule OK_le_err_OK [THEN iffD2])
   apply (drule OK_le_err_OK [THEN iffD2])
   apply (drule semilat_lub)
        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 blast
   apply (blast dest: plus_le_conv3)
  apply (erule subst)
  apply (blast intro: closedD)
  done
qed
```

26.4 semilat (err(Union AS))

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

lemma closed_err_Union_lift2I:
  "[| !A:AS. closed (err A) (lift2 f); AS ~= {};
      !A:AS.!B:AS. A~=B --> (!a:A.!b:B. a +_f b = Err) |]
  ==> closed (err(Union AS)) (lift2 f)"
apply (unfold closed_def err_def)
apply simp
apply clarify
apply simp
apply fast
done
```

If *AS = {}* the thm collapses to *order r* ∧ *closed {Err} f* ∧ *Err +_f Err = Err* which may not hold

```
lemma err_semilat_UnionI:
  "[| !A:AS. err_semilat(A, r, f); AS ~= {};
      !A:AS.!B:AS. A~=B --> (!a:A.!b:B. ~ a <=_r b & a +_f b = Err) |]
  ==> err_semilat(Union AS, r, f)"
apply (unfold semilat_def sl_def)
apply (simp add: closed_err_Union_lift2I)
apply (rule conjI)
 apply blast
apply (simp add: err_def)
apply (rule conjI)
 apply clarify
 apply (rename_tac A a u B b)
 apply (case_tac "A = B")
  apply simp
 apply simp
apply (rule conjI)
 apply clarify
 apply (rename_tac A a u B b)
 apply (case_tac "A = B")
  apply simp
 apply simp
apply clarify
apply (rename_tac A ya yb B yd z C c a b)
apply (case_tac "A = B")
 apply (case_tac "A = C")
  apply simp
 apply (rotate_tac -1)
 apply simp
apply (rotate_tac -1)
apply (case_tac "B = C")
 apply simp
apply (rotate_tac -1)
apply simp
done

end
```

27 More about Options

theory *Opt = Err:*

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

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

 sup :: "'a ebinop => 'a option ebinop"
"sup f o1 o2 ==
 case o1 of None => OK o2 | Some x => (case o2 of None => OK o1
     | Some y => (case f x y of Err => Err | OK z => OK (Some z)))"

 esl :: "'a esl => 'a option esl"
"esl == %(A,r,f). (opt A, le r, sup f)"
```

lemma *unfold_le_opt:*
```
  "o1 <=_(le r) o2 =
  (case o2 of None => o1=None |
              Some y => (case o1 of None => True | Some x => x <=_r y))"
```
apply *(unfold lesub_def le_def)*
apply *(rule refl)*
done

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

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

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

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

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

lemma *Some_le [iff]:*

```
  "(Some x <=_(le r) ox) = (? y. ox = Some y & x <=_r y)"
apply (unfold lesub_def le_def)
apply (simp split: option.split)
done

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


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

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

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


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

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


lemma semilat_opt:
  "!!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
```

28 Products as Semilattices

theory *Product = Err:*

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

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

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

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

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

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

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

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

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

lemma *closed_lift2_sup:*
"[| closed (err A) (lift2 f); closed (err B) (lift2 g) |] ==>
closed (err(A<>B)) (lift2(sup f g))"*
apply *(unfold closed_def plussub_def lift2_def err_def sup_def)*
apply *(simp split: err.split)*
apply *blast*
done

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


lemma plus_eq_Err_conv [simp]:
  "[| x:A; y:A; semilat(err A, Err.le r, lift2 f) |]
  ==> (x +_f y = Err) = (~(? z:A. x <=_r z & y <=_r z))"
proof -
  have plus_le_conv2:
    "!!r f z. [| z : err A; semilat (err A, r, f); OK x : err A; OK y : err A;
                 OK x +_f OK y <=_r z|] ==> OK x <=_r z ∧ OK y <=_r z"
    by (rule plus_le_conv [THEN iffD1])
  case rule_context
  thus ?thesis
  apply (rule_tac iffI)
   apply clarify
   apply (drule OK_le_err_OK [THEN iffD2])
   apply (drule OK_le_err_OK [THEN iffD2])
   apply (drule semilat_lub)
        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: semilatDorderI 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: semilatDclosedI 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: semilatDorderI)
apply (blast dest: semilatDorderI)

apply (rule OK_le_err_OK [THEN iffD1])
apply (erule subst, subst OK_lift2_OK [symmetric], rule semilat_lub)
```

apply *simp*
apply *simp*
apply *simp*
apply *simp*
apply *simp*
apply *simp*

apply *(rule OK_le_err_OK [THEN iffD1])*
apply *(erule subst, subst OK_lift2_OK [symmetric], rule semilat_lub)*
apply *simp*
apply *simp*
apply *simp*
apply *simp*
apply *simp*
apply *simp*
done

end

29 Fixed Length Lists

theory *Listn = Err:*

constdefs

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

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

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

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

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

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

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

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

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

lemmas *[simp] = set_update_subsetI*

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

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

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

done

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

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

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

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

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

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

lemma *le_list_antisym:*
"[| order r; xs <=[r] ys; ys <=[r] xs |] ==> xs = ys"
apply *(unfold unfold_lesub_list)*
apply *(simp add: Listn.le_def list_all2_conv_all_nth)*
apply *(rule nth_equalityI)*
apply *blast*
apply *clarify*
apply *simp*
apply *(blast intro: order_antisym)*
done

lemma *order_listI [simp, intro!]:*
"order r ==> order(Listn.le r)"

```
apply (subst order_def)
apply (blast intro: le_list_refl le_list_trans le_list_antisym
             dest: order_refl)
done


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

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

lemma 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 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 plus_list_ub1 [rule_format]:
  "[| semilat(A,r,f); 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 plus_list_ub2:
  "[| semilat(A,r,f); 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 plus_list_lub [rule_format]:
  "semilat(A,r,f) ==> !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 list_update_incr [rule_format]:
  "[| semilat(A,r,f); x:A |] ==> set xs <= A -->
  (!i. i<size xs --> xs <=[r] xs[i := x +_f xs!i])"
apply (unfold unfold_lesub_list)
apply (simp add: Listn.le_def list_all2_conv_all_nth)
apply (induct xs)
 apply simp
apply (simp add: in_list_Suc_iff)
apply clarify
apply (simp add: nth_Cons split: nat.split)
done

lemma acc_le_listI [intro!]:
  "[| order r; acc r |] ==> acc(Listn.le r)"
apply (unfold acc_def)
apply (subgoal_tac
 "wf(UN n. {(ys,xs). size xs = n & size ys = n & xs <_(Listn.le r) ys})")
 apply (erule wf_subset)
 apply (blast intro: lesssub_list_impl_same_size)
apply (rule wf_UN)
 prefer 2
 apply clarify
 apply (rename_tac m n)
 apply (case_tac "m=n")
  apply simp
 apply (rule conjI)
  apply (fast intro!: equals0I dest: not_sym)
 apply (fast intro!: equals0I dest: not_sym)
apply clarify
apply (rename_tac n)
apply (induct_tac n)
 apply (simp add: lesssub_def cong: conj_cong)
apply (rename_tac k)
apply (simp add: wf_eq_minimal)
apply (simp (no_asm) add: length_Suc_conv cong: conj_cong)
apply clarify
apply (rename_tac M m)
apply (case_tac "? x xs. size xs = k & x#xs : M")
 prefer 2
 apply (erule thin_rl)
 apply (erule thin_rl)
```

```
 apply blast
apply (erule_tac x = "{a. ? xs. size xs = k & a#xs:M}" in allE)
apply (erule impE)
 apply blast
apply (thin_tac "? x xs. ?P x xs")
apply clarify
apply (rename_tac maxA xs)
apply (erule_tac x = "{ys. size ys = size xs & maxA#ys : M}" in allE)
apply (erule impE)
 apply blast
apply clarify
apply (thin_tac "m : M")
apply (thin_tac "maxA#xs : M")
apply (rule bexI)
 prefer 2
 apply assumption
apply clarify
apply simp
apply blast
done

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


lemma semilat_Listn_sl:
  "!!L. semilat L ==> semilat (Listn.sl n L)"
apply (unfold Listn.sl_def)
apply (simp (no_asm_simp) only: split_tupled_all)
apply (simp (no_asm) only: semilat_Def split_conv)
apply (rule conjI)
 apply simp
apply (rule conjI)
 apply (simp only: semilatDclosedI closed_listI)
apply (simp (no_asm) only: list_def)
apply (simp (no_asm_simp) add: plus_list_ub1 plus_list_ub2 plus_list_lub)
done


lemma 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)
apply blast
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: semilatDclosedI closed_lift2_sup)
apply (rule conjI)
 apply (drule semilatDorderI)
 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
```

30 The Java Type System as Semilattice

theory *JType = WellForm + Err:*

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

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

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

  sup :: "'c prog => ty => ty => ty err"
  "sup G T1 T2 ==
  case T1 of PrimT P1 => (case T2 of PrimT P2 =>
                          (if P1 = P2 then OK (PrimT P1) else Err) | RefT R => Err)
           | RefT R1 => (case T2 of PrimT P => Err | RefT R2 =>
  (case R1 of NullT => (case R2 of NullT => OK NT | ClassT C => OK (Class C))
            | ClassT C => (case R2 of NullT => OK (Class C)
                                    | ClassT D => OK (Class (exec_lub (subcls1 G) (super G) C D)))))"

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

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

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

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

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

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

lemma *is_tyI:*
```
  "[| is_type G T; wf_prog wf_mb G |] ==> is_ty G T"
```
by *(auto simp add: is_ty_def intro: subcls_C_Object
          split: ty.splits ref_ty.splits)*

lemma *is_type_conv:*
```
  "wf_prog wf_mb G ==> is_type G T = is_ty G T"
```
proof
  assume *"is_type G T" "wf_prog wf_mb G"*
  thus *"is_ty G T"*
    by *(rule is_tyI)*
next
  assume *wf: "wf_prog wf_mb G"* **and**

```
        ty: "is_ty G T"

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

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

lemma wf_converse_subcls1_impl_acc_subtype:
  "wf ((subcls1 G)^-1) ==> acc (subtype G)"
apply (unfold acc_def lesssub_def)
apply (drule_tac p = "(subcls1 G)^-1 - Id" in wf_subset)
 apply blast
apply (drule wf_trancl)
apply (simp add: wf_eq_minimal)
apply clarify
apply (unfold lesub_def subtype_def)
apply (rename_tac M T)
apply (case_tac "EX C. Class C : M")
 prefer 2
 apply (case_tac T)
  apply (fastsimp simp add: PrimT_PrimT2)
 apply simp
 apply (subgoal_tac "ref_ty = NullT")
  apply simp
```

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

lemma closed_err_types:
  "[| wf_prog wf_mb G; single_valued (subcls1 G); acyclic (subcls1 G) |]
  ==> closed (err (types G)) (lift2 (sup G))"
  apply (unfold closed_def plussub_def lift2_def sup_def)
  apply (auto split: err.split)
  apply (drule is_tyI, assumption)
  apply (auto simp add: is_ty_def is_type_conv simp del: is_type.simps
              split: ty.split ref_ty.split)
  apply (blast dest!: is_lub_exec_lub is_lubD is_ubD intro!: is_ubI superI)
  done


lemma sup_subtype_greater:
  "[| wf_prog wf_mb G; single_valued (subcls1 G); acyclic (subcls1 G);
      is_type G t1; is_type G t2; sup G t1 t2 = OK s |]
  ==> subtype G t1 s ∧ subtype G t2 s"
proof -
  assume wf_prog:       "wf_prog wf_mb G"
  assume single_valued: "single_valued (subcls1 G)"
  assume acyclic:       "acyclic (subcls1 G)"

  { fix c1 c2
    assume is_class: "is_class G c1" "is_class G c2"
```

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

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

lemma sup_subtype_smallest:
  "[| wf_prog wf_mb G; single_valued (subcls1 G); acyclic (subcls1 G);
      is_type G a; is_type G b; is_type G c;
      subtype G a c; subtype G b c; sup G a b = OK d |]
  ==> subtype G d c"
proof -
  assume wf_prog:       "wf_prog wf_mb G"
  assume single_valued: "single_valued (subcls1 G)"
  assume acyclic:       "acyclic (subcls1 G)"

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

```
      by blast
  } note this [intro]

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

  assume "is_type G a" "is_type G b" "is_type G c"
         "subtype G a c" "subtype G b c" "sup G a b = OK d"
  thus ?thesis
    by (auto simp add: subtype_def sup_def
             split: ty.split_asm ref_ty.split_asm split_if_asm)
qed

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

lemma err_semilat_JType_esl_lemma:
  "[| wf_prog wf_mb G; single_valued (subcls1 G); acyclic (subcls1 G) |]
  ==> err_semilat (esl G)"
proof -
  assume wf_prog:   "wf_prog wf_mb G"
  assume single_valued: "single_valued (subcls1 G)"
  assume acyclic:   "acyclic (subcls1 G)"

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

  from wf_prog single_valued acyclic
  have
    "(∀x∈err (types G). ∀y∈err (types G). x <=_(le (subtype G)) x +_(lift2 (sup G)) y) ∧

     (∀x∈err (types G). ∀y∈err (types G). y <=_(le (subtype G)) x +_(lift2 (sup G)) y)"
    by (auto simp add: lesub_def plussub_def le_def lift2_def sup_subtype_greater split: err.split)

  moreover

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

  ultimately

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

lemma single_valued_subcls1:
```

```
  "wf_prog wf_mb G ==> single_valued (subcls1 G)"
  by (auto simp add: wf_prog_def unique_def single_valued_def
    intro: subcls1I elim!: subcls1.elims)

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

end
```

31 The JVM Type System as Semilattice

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

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

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

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

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

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

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

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

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

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

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

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

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

```
  sup_state     :: "['code prog,state_type,state_type] => bool"
                   ("_ ⊢ _ <=s _" [71,71] 70)
  sup_state_opt :: "['code prog,state_type option,state_type option] => bool"
                   ("_ ⊢ _ <=' _" [71,71] 70)


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


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

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

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

lemma zip_map [rule_format]:
  "∀a. length a = length b -->
  zip (map f a) (map g b) = map (λ(x,y). (f x, g y)) (zip a b)"
  apply (induct b)
   apply simp
  apply clarsimp
  apply (case_tac aa)
  apply simp+
  done

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

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

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

  { fix x' y'
    assume "length a = length b"
           "(x',y') ∈ set (zip (map OK a) (map OK b))"
    then
    obtain x y where OK:
```

```
      "x' = OK x" "y' = OK y" "(x,y) ∈ set (zip a b)"
      by (auto simp add: zip_map)
    with le
    have "subtype G x y"
      by (simp add: list_all2_def Ball_def)
    with OK
    have "G ⊢ x' <=o y'"
      by (simp add: sup_ty_opt_def)
  }

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

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


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


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

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

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

theorem sup_loc_refl [simp]:
  "G ⊢ t <=l t"
  by (simp add: sup_loc_def)

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

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


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

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

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

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

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

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

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

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


theorem sup_loc_length:
  "G ⊢ a <=l b ==> length a = length b"
proof -
  assume G: "G ⊢ a <=l b"
  have "∀b. (G ⊢ a <=l b) --> length a = length b"
    by (induct a, auto)
  with G
  show ?thesis by blast
qed

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

    fix x xs assume IH: "?P xs"

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

```
    qed
  qed
  with a
  show ?thesis by blast
qed

theorem all_nth_sup_loc:
  "∀b. length a = length b --> (∀ n. n < length a --> (G ⊢ (a!n) <=o (b!n)))
  --> (G ⊢ a <=l b)" (is "?P a")
proof (induct a)
  show "?P []" by simp

  fix l ls assume IH: "?P ls"

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

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

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

    with f b l IH
    show "G ⊢ (l # ls) <=l b"
      by auto
  qed
qed


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

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

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

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

    fix l ls assume IH: "?P ls"

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

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

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

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

  thus ?thesis by blast
qed


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

  fix l ls assume IH: "?P ls"
  show "?P (l#ls)"
  proof (intro strip)
    fix n y
    assume "G ⊢a <=o b" "G ⊢ (l # ls) <=l y" "n < length y"
    with IH
    show "G ⊢ (l # ls)[n := a] <=l y[n := b]"
```

```
    by - (cases n, auto simp add: sup_loc_Cons2 list_all2_Cons1)
  qed
qed


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

theorem sup_state_append_snd:
  "length a = length b ==>
  (G ⊢ (i,a@x) <=s (j,b@y)) = ((G ⊢ (i,a) <=s (j,b)) ∧ (G ⊢ (i,x) <=s (j,y)))"
  by (auto simp add: sup_state_def stk_convert lesub_def Product.le_def sup_loc_append)

theorem sup_state_append_fst:
  "length a = length b ==>
  (G ⊢ (a@x,i) <=s (b@y,j)) = ((G ⊢ (a,i) <=s (b,j)) ∧ (G ⊢ (x,i) <=s (y,j)))"
  by (auto simp add: sup_state_def stk_convert lesub_def Product.le_def sup_loc_append)

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

theorem sup_state_Cons2:
  "(G ⊢ (xt, a) <=s (y#yt, b)) =
   (∃x xt'. xt=x#xt' ∧ (G ⊢ x ≼ y) ∧ (G ⊢ (xt',a) <=s (yt,b)))"
  by (auto simp add: sup_state_def stk_convert lesub_def Product.le_def map_eq_Cons sup_loc_Cons2)

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

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


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

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

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

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

lemma sup_state_opt_Some_any:
```

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


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

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

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

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


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

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

end
```

32 Effect of Instructions on the State Type

theory *Effect = JVMType + JVMExceptions:*

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

Program counter of successor instructions:

consts
succs :: "instr => p_count => p_count list"
primrec
```
"succs (Load idx) pc         = [pc+1]"
"succs (Store idx) pc        = [pc+1]"
"succs (LitPush v) pc        = [pc+1]"
"succs (Getfield F C) pc     = [pc+1]"
"succs (Putfield F C) pc     = [pc+1]"
"succs (New C) pc            = [pc+1]"
"succs (Checkcast C) pc      = [pc+1]"
"succs Pop pc                = [pc+1]"
"succs Dup pc                = [pc+1]"
"succs Dup_x1 pc             = [pc+1]"
"succs Dup_x2 pc             = [pc+1]"
"succs Swap pc               = [pc+1]"
"succs IAdd pc               = [pc+1]"
"succs (Ifcmpeq b) pc        = [pc+1, nat (int pc + b)]"
"succs (Goto b) pc           = [nat (int pc + b)]"
"succs Return pc             = [pc]"
"succs (Invoke C mn fpTs) pc = [pc+1]"
"succs Throw pc              = [pc]"
```

Effect of instruction on the state type:

consts
eff' :: "instr × jvm_prog × state_type => state_type"

recdef *eff' "{}"*
```
"eff' (Load idx,  G, (ST, LT))          = (ok_val (LT ! idx) # ST, LT)"
"eff' (Store idx, G, (ts#ST, LT))       = (ST, LT[idx:= OK ts])"
"eff' (LitPush v, G, (ST, LT))           = (the (typeof (λv. None) v) # ST, LT)"
"eff' (Getfield F C, G, (oT#ST, LT))    = (snd (the (field (G,C) F)) # ST, LT)"
"eff' (Putfield F C, G, (vT#oT#ST, LT)) = (ST,LT)"
"eff' (New C, G, (ST,LT))               = (Class C # ST, LT)"
"eff' (Checkcast C, G, (RefT rt#ST,LT)) = (Class C # ST,LT)"
"eff' (Pop, G, (ts#ST,LT))              = (ST,LT)"
"eff' (Dup, G, (ts#ST,LT))              = (ts#ts#ST,LT)"
"eff' (Dup_x1, G, (ts1#ts2#ST,LT))      = (ts1#ts2#ts1#ST,LT)"
"eff' (Dup_x2, G, (ts1#ts2#ts3#ST,LT))  = (ts1#ts2#ts3#ts1#ST,LT)"
"eff' (Swap, G, (ts1#ts2#ST,LT))        = (ts2#ts1#ST,LT)"
"eff' (IAdd, G, (PrimT Integer#PrimT Integer#ST,LT))
                                         = (PrimT Integer#ST,LT)"
"eff' (Ifcmpeq b, G, (ts1#ts2#ST,LT))   = (ST,LT)"
"eff' (Goto b, G, s)                    = s"
```
— Return has no successor instruction in the same method
```
"eff' (Return, G, s)                    = s"
```

— Throw always terminates abruptly

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

consts

```
  match_any :: "jvm_prog ⇒ p_count ⇒ exception_table ⇒ cname list"
```

primrec

```
  "match_any G pc [] = []"
  "match_any G pc (e#es) = (let (start_pc, end_pc, handler_pc, catch_type) = e;
                                es' = match_any G pc es
                            in
                            if start_pc <= pc ∧ pc < end_pc then catch_type#es' else es')"
```

consts

```
  xcpt_names :: "instr × jvm_prog × p_count × exception_table => cname list"
```

recdef *xcpt_names "{}"*

```
  "xcpt_names (Getfield F C, G, pc, et) = [Xcpt NullPointer]"
  "xcpt_names (Putfield F C, G, pc, et) = [Xcpt NullPointer]"
  "xcpt_names (New C, G, pc, et)        = [Xcpt OutOfMemory]"
  "xcpt_names (Checkcast C, G, pc, et)  = [Xcpt ClassCast]"
  "xcpt_names (Throw, G, pc, et)        = match_any G pc et"
  "xcpt_names (Invoke C m p, G, pc, et) = match_any G pc et"
  "xcpt_names (i, G, pc, et)            = []"
```

constdefs

```
  xcpt_eff :: "instr ⇒ jvm_prog ⇒ p_count ⇒ state_type option ⇒ exception_table ⇒ succ_ty
  "xcpt_eff i G pc s et ==
   map (λC. (the (match_exception_table G C pc et), case s of None ⇒ None | Some s' ⇒ Some
([Class C], snd s')))
       (xcpt_names (i,G,pc,et))"

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

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

constdefs

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

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

lemma *isPrimT [simp]:*

```
  "isPrimT T = (∃T'. T = PrimT T')" by (simp add: isPrimT_def split: ty.splits)
```

lemma *isRefT [simp]:*
"isRefT T = (∃T'. T = RefT T')" **by** *(simp add: isRefT_def split: ty.splits)*

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

Conditions under which eff is applicable:

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

recdef *app' "{}"*
```
"app' (Load idx, G, maxs, rT, s)                  = (idx < length (snd s) ∧
                                                     (snd s) ! idx ≠ Err ∧
                                                     length (fst s) < maxs)"
"app' (Store idx, G, maxs, rT, (ts#ST, LT))       = (idx < length LT)"
"app' (LitPush v, G, maxs, rT, s)                  = (length (fst s) < maxs ∧ typeof (λt. None)
v ≠ None)"
"app' (Getfield F C, G, 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, 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, maxs, rT, s)                      = (is_class G C ∧ length (fst s) < maxs)"
"app' (Checkcast C, G, maxs, rT, (RefT rt#ST,LT)) = (is_class G C)"
"app' (Pop, G, maxs, rT, (ts#ST,LT))               = True"
"app' (Dup, G, maxs, rT, (ts#ST,LT))               = (1+length ST < maxs)"
"app' (Dup_x1, G, maxs, rT, (ts1#ts2#ST,LT))       = (2+length ST < maxs)"
"app' (Dup_x2, G, maxs, rT, (ts1#ts2#ts3#ST,LT))   = (3+length ST < maxs)"
"app' (Swap, G, maxs, rT, (ts1#ts2#ST,LT))         = True"
"app' (IAdd, G, maxs, rT, (PrimT Integer#PrimT Integer#ST,LT))
                                                  = True"
"app' (Ifcmpeq b, G, maxs, rT, (ts#ts'#ST,LT))     = (isPrimT ts ∧ ts' = ts ∨
                                                     isRefT ts ∧ isRefT ts')"
"app' (Goto b, G, maxs, rT, s)                     = True"
"app' (Return, G, maxs, rT, (T#ST,LT))             = (G ⊢ T ⪯ rT)"
"app' (Throw, G, maxs, rT, (T#ST,LT))              = isRefT T"

"app' (Invoke C mn fpTs, G, 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,maxs,rT,s)                              = False"
```

constdefs

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

  app :: "instr => jvm_prog => nat => ty => nat => exception_table => state_type option => bool
  "app i G maxs rT pc et s == case s of None => True | Some t => app' (i,G,maxs,rT,t) ∧ xcpt_ap
i G pc et"
```

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 ∧ typeo
(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) ∧ is_class G (Xcpt NullPointer))"
  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' ∧ is_class G (Xcpt NullPointer))
  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 ∧ is_class G (Xcpt OutOfMemory))"
  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 ∧ is_class G (Xcpt ClassCast))"
  by (cases s, cases "fst s", simp add: app_def) (cases "hd (fst s)", auto)
```

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

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

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

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

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

lemma *appIAdd[simp]:*
```
"app IAdd G maxs rT pc et (Some s) = (∃ ST LT. s = (PrimT Integer#PrimT Integer#ST,LT))"
  (is "?app s = ?P s")
```

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


lemma appIfcmpeq[simp]:
"app (Ifcmpeq b) G maxs rT pc et (Some s) = (∃ts1 ts2 ST LT. s = (ts1#ts2#ST,LT) ∧
 ((∃ 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 branch) G maxs rT pc et (Some s) = True"
  by simp

lemma appThrow[simp]:
  "app Throw G maxs rT pc et (Some s) =
  (∃T ST LT r. s=(T#ST,LT) ∧ T = RefT r ∧ (∀C ∈ set (match_any G pc et). is_class G C))"
  by (cases s, cases "2 < length (fst s)", auto dest: 1 2)

lemma appInvoke[simp]:
"app (Invoke C mn fpTs) G maxs rT pc et (Some s) = (∃apTs X ST LT mD' rT' b'.
  s = ((rev apTs) @ (X # ST), LT) ∧ length apTs = length fpTs ∧ is_class G C ∧
  G ⊢ X ≼ Class C ∧ (∀(aT,fT)∈set(zip apTs fpTs). G ⊢ aT ≼ fT) ∧
  method (G,C) (mn,fpTs) = Some (mD', rT', b') ∧
  (∀C ∈ set (match_any G pc et). is_class G C))" (is "?app s = ?P s")
```

```
proof (cases (open) s)
  note list_all2_def [simp]
  case Pair
  have "?app (a,b) ==> ?P (a,b)"
  proof -
    assume app: "?app (a,b)"
    hence "a = (rev (rev (take (length fpTs) a))) @ (drop (length fpTs) a) ∧
          length fpTs < length a" (is "?a ∧ ?l")
      by (auto simp add: app_def)
    hence "?a ∧ 0 < length (drop (length fpTs) a)" (is "?a ∧ ?l")
      by auto
    hence "?a ∧ ?l ∧ length (rev (take (length fpTs) a)) = length fpTs"
      by (auto simp add: min_def)
    hence "∃apTs ST. a = rev apTs @ ST ∧ length apTs = length fpTs ∧ 0 < length ST"
      by blast
    hence "∃apTs ST. a = rev apTs @ ST ∧ length apTs = length fpTs ∧ ST ≠ []"
      by blast
    hence "∃apTs ST. a = rev apTs @ ST ∧ length apTs = length fpTs ∧
          (∃X ST'. ST = X#ST')"
      by (simp add: neq_Nil_conv)
    hence "∃apTs X ST. a = rev apTs @ X # ST ∧ length apTs = length fpTs"
      by blast
    with app
    show ?thesis by (unfold app_def, clarsimp) blast
  qed
  with Pair
  have "?app s ⟹ ?P s" by (simp only:)
  moreover
  have "?P s ⟹ ?app s" by (unfold app_def) (clarsimp simp add: min_def)
  ultimately
  show ?thesis by (rule iffI)
qed

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

some helpers to make the specification directly executable:

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

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

lemmas [simp del] = app_def xcpt_app_def

end
```

33 The Bytecode Verifier

theory *BVSpec = Effect:*

This theory contains a specification of the BV. The specification describes correct typings of method bodies; it corresponds to type *checking*.

constdefs

```
  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')"

  wt_start :: "[jvm_prog,cname,ty list,nat,method_type] => bool"
  "wt_start G C pTs mxl phi ==
  G ⊢ Some ([],(OK (Class C))#((map OK pTs))@(replicate mxl Err)) <=' phi!0"

  wt_method :: "[jvm_prog,cname,ty list,ty,nat,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 ∧ wt_start G C pTs mxl phi ∧
  (∀pc. pc<max_pc --> wt_instr (ins ! pc) G rT phi mxs max_pc et pc)"

  wt_jvm_prog :: "[jvm_prog,prog_type] => bool"
  "wt_jvm_prog G phi ==
  wf_prog (λG C (sig,rT,(maxs,maxl,b,et)).
           wt_method G C (snd sig) rT maxs maxl b et (phi C sig)) G"
```

lemma *wt_jvm_progD:*

```
  "wt_jvm_prog G phi ==> (∃wt. wf_prog wt G)"
  by (unfold wt_jvm_prog_def, blast)
```

lemma *wt_jvm_prog_impl_wt_instr:*

```
  "[| wt_jvm_prog G phi; 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)
```

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

34 BV Type Safety Invariant

theory *Correct = BVSpec + JVMExec:*

constdefs

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

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

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

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

consts

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

primrec

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

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

constdefs

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

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


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

34.1 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)
```

34.2 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 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
```

34.3 approx-stk

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

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

lemma *approx_stk_sup_heap:*
"⟦ approx_stk G hp stk ST; hp ≤| hp' ⟧ ⟹ approx_stk G hp' stk ST"
by *(auto intro: approx_loc_sup_heap simp add: approx_stk_def)*

lemma *approx_stk_widen:*
"⟦ approx_stk G hp stk ST; G ⊢ map OK ST <=l map OK ST'; wf_prog wt G ⟧
⟹ approx_stk G hp stk ST'"
by *(auto elim: approx_loc_widen simp add: approx_stk_def)*

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

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

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

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

lemma *approx_stk_all_widen:*
"⟦ approx_stk G hp stk ST; ∀x ∈ set (zip ST ST'). x ∈ widen G; length ST = length ST'; wf_p
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

34.4 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

34.5 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

34.6 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 *preallocated_newref:*
"⟦ *hp oref = None; preallocated hp* ⟧ ⟹ *preallocated (hp(oref↦obj))*"
by *(unfold preallocated_def) auto*

34.7 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 ⟶
  G,hp ⊢ obj √
  ⟶ 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
```

35 BV Type Safety Proof

theory *BVSpecTypeSafe = Correct:*

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

35.1 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

35.2 Exception Handling

Exceptions don't touch anything except the stack:

lemma *exec_instr_xcpt:*
"(fst (exec_instr i G hp stk vars Cl sig pc frs) = Some xcp)
= (∃stk'. exec_instr i G hp stk vars Cl sig pc frs =
(Some xcp, hp, (stk', vars, Cl, sig, pc)#frs))"
by *(cases i, auto simp add: split_beta split: split_if_asm)*

Relates *match_any* from the Bytecode Verifier with *match_exception_table* from the operational semantics:

lemma *in_match_any:*
"match_exception_table G xcpt pc et = Some pc' ==>
∃C. C ∈ set (match_any G pc et) ∧ G ⊢ xcpt ⪯C C ∧
match_exception_table G C pc et = Some pc'"
*(***is** *"PROP ?P et"* **is** *"?match et ==> ?match_any et")*
proof *(induct et)*
show *"PROP ?P []"*
by *simp*

fix *e es*
assume *IH: "PROP ?P es"*
assume *match: "?match (e#es)"*

obtain *start_pc end_pc handler_pc catch_type* **where**

```
    e [simp]: "e = (start_pc, end_pc, handler_pc, catch_type)"
    by (cases e)

  from IH match
  show "?match_any (e#es)"
  proof (cases "match_exception_entry G xcpt pc e")
    case False
    with match
    have "match_exception_table G xcpt pc es = Some pc'" by simp
    with IH
    obtain C where
      set: "C ∈ set (match_any G pc es)" and
      C:   "G ⊢ xcpt ⪯C C" and
      m:   "match_exception_table G C pc es = Some pc'" by blast

    from set
    have "C ∈ set (match_any G pc (e#es))" by simp
    moreover
    from False C
    have "¬ match_exception_entry G C pc e"
      by - (erule contrapos_nn,
            auto simp add: match_exception_entry_def elim: rtrancl_trans)
    with m
    have "match_exception_table G C pc (e#es) = Some pc'" by simp
    moreover note C
    ultimately
    show ?thesis by blast
  next
    case True with match
    have "match_exception_entry G catch_type pc e"
      by (simp add: match_exception_entry_def)
    moreover
    from True match
    obtain
      "start_pc ≤ pc"
      "pc < end_pc"
      "G ⊢ xcpt ⪯C catch_type"
      "handler_pc = pc'"
      by (simp add: match_exception_entry_def)
    ultimately
    show ?thesis by auto
  qed
qed
```

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
```

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)
  next
    case Throw with xcpt wt
    show ?thesis
      by (auto simp add: wt_instr_def correct_state_def correct_frame_def
               dest: non_npD)
  qed auto
qed
```

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" by simp
```

```
    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) blast
    note phi'
    moreover
    { from xcp prehp
      have "G,hp ⊢ xcp ::≼ Class (Xcpt OutOfMemory)"
        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 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" by simp
    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) blast
    note phi'
    moreover
    { from xcp prehp
      have "G,hp ⊢ xcp ::≼ Class (Xcpt NullPointer)"
        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 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" by simp
    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) blast
    note phi'
    moreover
    { from xcp prehp
      have "G,hp ⊢ xcp ::≼ Class (Xcpt NullPointer)"
        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 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" by simp
    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) blast
    note phi'
    moreover
    { from xcp prehp
      have "G,hp ⊢ xcp ::≼ Class (Xcpt ClassCast)"
        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 Invoke
    with phi_pc eff
    have
```

```
    "∀D∈set (match_any G pc et).
    the (?m D) < length ins ∧ G ⊢ Some ([Class D], LT) <=' phi C sig!the (?m D)"
    by (simp add: xcpt_eff_def)
  moreover
  from some_handler
  obtain D where
    "D ∈ set (match_any G pc et)" and
    D: "G ⊢ cname_of hp xcp ⪯C D" and
    "?m D = Some handler"
    by (blast dest: in_match_any)
  ultimately
  obtain
    pc': "handler < length ins" and
    "G ⊢ Some ([Class D], LT) <=' phi C sig ! handler"
    by auto
  then
  obtain ST' LT' where
    phi': "phi C sig ! handler = Some (ST', LT')" and
    less: "G ⊢ ([Class D], LT) <=s (ST', LT')"
    by auto
  from xp wt correct
  obtain addr T where
    xcp: "xcp = Addr addr" "hp addr = Some T"
    by (blast dest: exec_instr_xcpt_hp)
  note phi'
  moreover
  { from xcp D
    have "G,hp ⊢ xcp ::≼ Class D"
      by (simp add: conf_def obj_ty_def)
    moreover
    from wf less loc
    have "approx_loc G hp loc LT'"
      by (simp add: sup_state_conv) blast
    moreover
    note wf less pc' len
    ultimately
    have "correct_frame G hp (ST',LT') maxl ins ?f'"
      by (unfold correct_frame_def) (auto simp add: sup_state_conv
          approx_stk_def approx_val_def split: err.split elim: conf_widen)
  }
  ultimately
  show ?thesis by (rule that)
next
  case Throw
  with phi_pc eff
  have
    "∀D∈set (match_any G pc et).
    the (?m D) < length ins ∧ G ⊢ Some ([Class D], LT) <=' phi C sig!the (?m D)"
    by (simp add: xcpt_eff_def)
  moreover
  from some_handler
  obtain D where
    "D ∈ set (match_any G pc et)" and
    D: "G ⊢ cname_of hp xcp ⪯C D" and
    "?m D = Some handler"
    by (blast dest: in_match_any)
  ultimately
  obtain
    pc': "handler < length ins" and
```

```
        "G ⊢ Some ([Class D], LT) <=' phi C sig ! handler"
        by auto
      then
      obtain ST' LT' where
        phi': "phi C sig ! handler = Some (ST', LT')" and
        less: "G ⊢ ([Class D], LT) <=s (ST', LT')"
        by auto
      from xp wt correct
      obtain addr T where
        xcp: "xcp = Addr addr" "hp addr = Some T"
        by (blast dest: exec_instr_xcpt_hp)
      note phi'
      moreover
      { from xcp D
        have "G,hp ⊢ xcp ::≼ Class D"
          by (simp add: conf_def obj_ty_def)
        moreover
        from wf less loc
        have "approx_loc G hp loc LT'"
          by (simp add: sup_state_conv) blast
        moreover
        note wf less pc' len
        ultimately
        have "correct_frame G hp (ST',LT') maxl ins ?f'"
          by (unfold correct_frame_def) (auto simp add: sup_state_conv
              approx_stk_def approx_val_def split: err.split elim: conf_widen)
      }
      ultimately
      show ?thesis by (rule that)
    qed (insert xp', auto) — the other instructions don't generate exceptions

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

35.3 Single Instructions

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

```
lemmas [iff] = not_Err_eq

lemma Load_correct:
"[| wf_prog wt G;
    method (G,C) sig = Some (C,rT,maxs,maxl,ins,et);
    ins!pc = Load idx;
    wt_instr (ins!pc) G rT (phi C sig) maxs (length ins) et pc;
    Some state' = exec (G, None, hp, (stk,loc,C,sig,pc)#frs);
    G,phi ⊢JVM (None, hp, (stk,loc,C,sig,pc)#frs)√ |]
==> G,phi ⊢JVM state'√"
apply (clarsimp simp add: defs1 map_eq_Cons)
apply (blast intro: approx_loc_imp_approx_val_sup)
done
```

lemma *Store_correct:*
```
"[| wf_prog wt G;
  method (G,C) sig = Some (C,rT,maxs,maxl,ins,et);
  ins!pc = Store idx;
  wt_instr (ins!pc) G rT (phi C sig) maxs (length ins) et pc;
  Some state' = exec (G, None, hp, (stk,loc,C,sig,pc)#frs);
  G,phi ⊢JVM (None, hp, (stk,loc,C,sig,pc)#frs)√ |]
==> G,phi ⊢JVM state'√"
```
apply *(clarsimp simp add: defs1 map_eq_Cons)*
apply *(blast intro: approx_loc_subst)*
done

lemma *LitPush_correct:*
```
"[| wf_prog wt G;
    method (G,C) sig = Some (C,rT,maxs,maxl,ins,et);
    ins!pc = LitPush v;
    wt_instr (ins!pc) G rT (phi C sig) maxs (length ins) et pc;
    Some state' = exec (G, None, hp, (stk,loc,C,sig,pc)#frs);
    G,phi ⊢JVM (None, hp, (stk,loc,C,sig,pc)#frs)√ |]
==> G,phi ⊢JVM state'√"
```
apply *(clarsimp simp add: defs1 sup_PTS_eq map_eq_Cons)*
apply *(blast dest: conf_litval intro: conf_widen)*
done

lemma *Cast_conf2:*
```
  "[| wf_prog ok G; G,h⊢v::≼RefT rt; cast_ok G C h v;
      G⊢Class C≼T; is_class G C|]
  ==> G,h⊢v::≼T"
```
apply *(unfold cast_ok_def)*
apply *(frule widen_Class)*
apply *(elim exE disjE)*
apply *(simp add: null)*
apply *(clarsimp simp add: conf_def obj_ty_def)*
apply *(cases v)*
apply *(auto intro: rtrancl_trans)*
done

lemmas *defs1 = 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: defs1 wt_jvm_prog_def map_eq_Cons split: split_if_asm)*

```
apply (blast intro: Cast_conf2)
done


lemma Getfield_correct:
"[| wt_jvm_prog G phi;
  method (G,C) sig = Some (C,rT,maxs,maxl,ins,et);
  ins!pc = Getfield F D;
  wt_instr (ins!pc) G rT (phi C sig) maxs (length ins) et pc;
  Some state' = exec (G, None, hp, (stk,loc,C,sig,pc)#frs) ;
  G,phi ⊢JVM (None, hp, (stk,loc,C,sig,pc)#frs)√;
  fst (exec_instr (ins!pc) G hp stk loc C sig pc frs) = None |]
==> G,phi ⊢JVM state'√"
apply (clarsimp simp add: defs1 map_eq_Cons wt_jvm_prog_def
                split: option.split split_if_asm)
apply (frule conf_widen)
apply assumption+
apply (drule conf_RefTD)
apply (clarsimp simp add: defs1)
apply (rule conjI)
 apply (drule widen_cfs_fields)
 apply assumption+
 apply (erule conf_widen)
 prefer 2
  apply assumption
 apply (simp add: hconf_def oconf_def lconf_def)
 apply (elim allE)
 apply (erule impE, assumption)
 apply simp
 apply (elim allE)
 apply (erule impE, assumption)
 apply clarsimp
apply blast
done


lemma Putfield_correct:
"[| wf_prog wt G;
  method (G,C) sig = Some (C,rT,maxs,maxl,ins,et);
  ins!pc = Putfield F D;
  wt_instr (ins!pc) G rT (phi C sig) maxs (length ins) et pc;
  Some state' = exec (G, None, hp, (stk,loc,C,sig,pc)#frs) ;
  G,phi ⊢JVM (None, hp, (stk,loc,C,sig,pc)#frs)√;
  fst (exec_instr (ins!pc) G hp stk loc C sig pc frs) = None |]
==> G,phi ⊢JVM state'√"
apply (clarsimp simp add: defs1 split_beta split: option.split List.split split_if_asm)
apply (frule conf_widen)
prefer 2
  apply assumption
 apply assumption
apply (drule conf_RefTD)
apply clarsimp
```

```
apply (blast
       intro:
         hext_upd_obj approx_stk_sup_heap
         approx_loc_sup_heap
         hconf_field_update
         preallocated_field_update
         correct_frames_field_update conf_widen
       dest:
         widen_cfs_fields)
done


lemma New_correct:
"[| wf_prog wt G;
  method (G,C) sig = Some (C,rT,maxs,maxl,ins,et);
  ins!pc = New X;
  wt_instr (ins!pc) G rT (phi C sig) maxs (length ins) et pc;
  Some state' = exec (G, None, hp, (stk,loc,C,sig,pc)#frs) ;
  G,phi ⊢JVM (None, hp, (stk,loc,C,sig,pc)#frs)√;
  fst (exec_instr (ins!pc) G hp stk loc C sig pc frs) = None |]
==> G,phi ⊢JVM state'√"
proof -
  assume wf:   "wf_prog wt G"
  assume meth: "method (G,C) sig = Some (C,rT,maxs,maxl,ins,et)"
  assume ins:  "ins!pc = New X"
  assume wt:   "wt_instr (ins!pc) G rT (phi C sig) maxs (length ins) et pc"
  assume exec: "Some state' = exec (G, None, hp, (stk,loc,C,sig,pc)#frs)"
  assume conf: "G,phi ⊢JVM (None, hp, (stk,loc,C,sig,pc)#frs)√"
  assume no_x: "fst (exec_instr (ins!pc) G hp stk loc C sig pc frs) = None"

  from ins conf meth
  obtain ST LT where
    heap_ok:    "G⊢h hp√" and
    prealloc:   "preallocated hp" and
    phi_pc:     "phi C sig!pc = Some (ST,LT)" and
    is_class_C: "is_class G C" and
    frame:      "correct_frame G hp (ST,LT) maxl ins (stk, loc, C, sig, pc)" and
    frames:     "correct_frames G hp phi rT sig frs"
    by (auto simp add: correct_state_def iff del: not_None_eq)

  from phi_pc ins wt
  obtain ST' LT' where
    is_class_X: "is_class G X" and
    maxs:       "length ST < maxs" and
    suc_pc:     "Suc pc < length ins" and
    phi_suc:    "phi C sig ! Suc pc = Some (ST', LT')" and
    less:       "G ⊢ (Class X # ST, LT) <=s (ST', LT')"
    by (unfold wt_instr_def eff_def norm_eff_def) auto

  obtain oref xp' where
    new_Addr: "new_Addr hp = (oref,xp')"
    by (cases "new_Addr hp")
```

```
  with ins no_x
  obtain hp: "hp oref = None" and "xp' = None"
    by (auto dest: new_AddrD simp add: raise_system_xcpt_def)

  with exec ins meth new_Addr
  have state':
    "state' = Norm (hp(oref↦(X, init_vars (fields (G, X)))),
              (Addr oref # stk, loc, C, sig, Suc pc) # frs)"
    (is "state' = Norm (?hp', ?f # frs)")
    by simp
  moreover
  from wf hp heap_ok is_class_X
  have hp': "G ⊢h ?hp' √"
    by - (rule hconf_newref, auto simp add: oconf_def dest: fields_is_type)
  moreover
  from hp
  have sup: "hp ≤| ?hp'" by (rule hext_new)
  from hp frame less suc_pc wf
  have "correct_frame G ?hp' (ST', LT') maxl ins ?f"
    apply (unfold correct_frame_def sup_state_conv)
    apply (clarsimp simp add: map_eq_Cons conf_def fun_upd_apply approx_val_def)
    apply (blast intro: approx_stk_sup_heap approx_loc_sup_heap sup)
    done
  moreover
  from hp frames wf heap_ok is_class_X
  have "correct_frames G ?hp' phi rT sig frs"
    by - (rule correct_frames_newref,
          auto simp add: oconf_def dest: fields_is_type)
  moreover
  from hp prealloc have "preallocated ?hp'" by (rule preallocated_newref)
  ultimately
  show ?thesis
    by (simp add: is_class_C meth phi_suc correct_state_def del: not_None_eq)
qed

lemmas [simp del] = split_paired_Ex


lemma Invoke_correct:
"[| wt_jvm_prog G phi;
  method (G,C) sig = Some (C,rT,maxs,maxl,ins,et);
  ins ! pc = Invoke C' mn pTs;
  wt_instr (ins!pc) G rT (phi C sig) maxs (length ins) et pc;
  Some state' = exec (G, None, hp, (stk,loc,C,sig,pc)#frs) ;
  G,phi ⊢JVM (None, hp, (stk,loc,C,sig,pc)#frs)√;
  fst (exec_instr (ins!pc) G hp stk loc C sig pc frs) = None |]
==> G,phi ⊢JVM state'√"
proof -
  assume wtprog: "wt_jvm_prog G phi"
  assume method: "method (G,C) sig = Some (C,rT,maxs,maxl,ins,et)"
  assume ins:    "ins ! pc = Invoke C' mn pTs"
  assume wti:    "wt_instr (ins!pc) G rT (phi C sig) maxs (length ins) et pc"
```

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

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

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

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

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

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

from s ins frame
obtain
  a_stk: "approx_stk G hp stk (rev apTs @ X # ST)" and
  a_loc: "approx_loc G hp loc LT" and
  suc_l: "length loc = Suc (length (snd sig) + maxl)"
  by (simp add: correct_frame_def)

from a_stk
obtain opTs stk' oX where
  opTs:   "approx_stk G hp opTs (rev apTs)" and
```

```
  oX:     "approx_val G hp oX (OK X)" and
  a_stk': "approx_stk G hp stk' ST" and
  stk':   "stk = opTs @ oX # stk'" and
  l_o:    "length opTs = length apTs"
          "length stk' = length ST"
  by - (drule approx_stk_append, auto)

from oX X_Ref
have oX_conf: "G,hp ⊢ oX ::≼ RefT T"
  by (simp add: approx_val_def)

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

with state' method ins no_xcp oX_conf
obtain ref where oX_Addr: "oX = Addr ref"
  by (auto simp add: raise_system_xcpt_def dest: conf_RefTD)

with oX_conf X_Ref
obtain obj D where
  loc:    "hp ref = Some obj" and
  obj_ty: "obj_ty obj = Class D" and
  D:      "G ⊢ Class D ≼ X"
  by (auto simp add: conf_def) blast

with X_Ref obtain X' where X': "X = Class X'"
  by (blast dest: widen_Class)

with X have X'_subcls: "G ⊢ X' ≼C C'"  by simp

with mC' wfprog
obtain D0 rT0 maxs0 maxl0 ins0 et0 where
  mX: "method (G, X') (mn, pTs) = Some (D0, rT0, maxs0, maxl0, ins0, et0)" "G⊢rT0≼rT"
  by (auto dest: subtype_widen_methd intro: that)

from X' D have D_subcls: "G ⊢ D ≼C X'" by simp

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

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

from is_class X'_subcls D_subcls
have is_class_D: "is_class G D" by (auto dest: subcls_is_class2)

with mD wfprog
obtain mD'':
  "method (G, D'') (mn, pTs) = Some (D'', rT', mxs', mxl', ins', et')"
  "is_class G D''"
```

```
  by (auto dest: method_in_md)

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

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

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

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

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

  have r: "approx_loc G hp (replicate mxl' arbitrary) (replicate mxl' Err)"
    by (simp add: approx_loc_def list_all2_def set_replicate_conv_if)

  from wfprog mD is_class_D
  have "G ⊢ Class D ≼ Class D''"
    by (auto dest: method_wf_mdecl)
  with obj_ty loc
  have a: "approx_val G hp (Addr ref) (OK (Class D''))"
    by (simp add: approx_val_def conf_def)

  from opTs w l l_o wfprog
  have "approx_stk G hp opTs (rev pTs)"
    by (auto elim!: approx_stk_all_widen simp add: zip_rev)
  hence "approx_stk G hp (rev opTs) pTs" by (subst approx_stk_rev)

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

  from this sup_loc wfprog
  have "approx_loc G hp ?lt LT0" by (rule approx_loc_widen)
  with start l_o l
  show ?thesis by (simp add: correct_frame_def)
```

```
  qed

  from state'_val heap_ok mD'' ins method phi_pc s X' l mX
       frames s' LT0 c_f is_class_C stk' oX_Addr frame prealloc
  show ?thesis by (simp add: correct_state_def) (intro exI conjI, blast)
qed

lemmas [simp del] = map_append

lemma Return_correct:
"[| wt_jvm_prog G phi;
  method (G,C) sig = Some (C,rT,maxs,maxl,ins,et);
  ins ! pc = Return;
  wt_instr (ins!pc) G rT (phi C sig) maxs (length ins) et pc;
  Some state' = exec (G, None, hp, (stk,loc,C,sig,pc)#frs) ;
  G,phi ⊢JVM (None, hp, (stk,loc,C,sig,pc)#frs)√ |]
==> G,phi ⊢JVM state'√"
proof -
  assume wt_prog: "wt_jvm_prog G phi"
  assume meth: "method (G,C) sig = Some (C,rT,maxs,maxl,ins,et)"
  assume ins: "ins ! pc = Return"
  assume wt: "wt_instr (ins!pc) G rT (phi C sig) maxs (length ins) et pc"
  assume s': "Some state' = exec (G, None, hp, (stk,loc,C,sig,pc)#frs)"
  assume correct: "G,phi ⊢JVM (None, hp, (stk,loc,C,sig,pc)#frs)√"

  from wt_prog
  obtain wfmb where wf: "wf_prog wfmb G" by (simp add: wt_jvm_prog_def)

  from meth ins s'
  have "frs = [] ⟹ ?thesis" by (simp add: correct_state_def)
  moreover
  { fix f frs'
    assume frs': "frs = f#frs'"
    moreover
    obtain stk' loc' C' sig' pc' where
      f: "f = (stk',loc',C',sig',pc')" by (cases f)
    moreover
    obtain mn pt where
      sig: "sig = (mn,pt)" by (cases sig)
    moreover
    note meth ins s'
    ultimately
    have state':
      "state' = (None,hp,(hd stk#(drop (1+length pt) stk'),loc',C',sig',pc'+1)#frs')"
      (is "state' = (None,hp,?f'#frs')")
      by simp

    from correct meth
    obtain ST LT where
      hp_ok:  "G ⊢h hp √" and
      alloc:  "preallocated hp" and
      phi_pc: "phi C sig ! pc = Some (ST, LT)" and
```

```
  frame:  "correct_frame G hp (ST, LT) maxl ins (stk,loc,C,sig,pc)" and
  frames: "correct_frames G hp phi rT sig frs"
  by (simp add: correct_state_def, clarify, blast)

from phi_pc ins wt
obtain T ST' where "ST = T # ST'" "G ⊢ T ⪯ rT"
  by (simp add: wt_instr_def) blast
with wf frame
have hd_stk: "G,hp ⊢ (hd stk) ::⪯ rT"
  by (auto simp add: correct_frame_def elim: conf_widen)

from f frs' frames sig
obtain apTs ST0' ST' LT' D D' D'' rT' rT'' maxs' maxl' ins' et' body where
  phi':   "phi C' sig' ! pc' = Some (ST',LT')" and
  class': "is_class G C'" and
  meth':  "method (G,C') sig' = Some (C',rT',maxs',maxl',ins',et')" and
  ins':   "ins' ! pc' = Invoke D' mn pt" and
  frame': "correct_frame G hp (ST', LT') maxl' ins' f" and
  frames':"correct_frames G hp phi rT' sig' frs'" and
  rT'':   "G ⊢ rT ⪯ rT''" and
  meth'': "method (G, D) sig = Some (D'', rT'', body)" and
  ST0':   "ST' = rev apTs @ Class D # ST0'" and
  len':   "length apTs = length pt"
  by clarsimp blast

from f frame'
obtain
  stk': "approx_stk G hp stk' ST'" and
  loc': "approx_loc G hp loc' LT'" and
  pc':  "pc' < length ins'" and
  lloc':"length loc' = Suc (length (snd sig') + maxl')"
  by (simp add: correct_frame_def)

from wt_prog class' meth' pc'
have "wt_instr (ins'!pc') G rT' (phi C' sig') maxs' (length ins') et' pc'"
  by (rule wt_jvm_prog_impl_wt_instr)
with ins' phi' sig
obtain apTs ST0 X ST'' LT'' body' rT0 mD where
  phi_suc: "phi C' sig' ! Suc pc' = Some (ST'', LT'')" and
  ST0:     "ST' = rev apTs @ X # ST0" and
  len:     "length apTs = length pt" and
  less:    "G ⊢ (rT0 # ST0, LT') <=s (ST'', LT'')" and
  methD':  "method (G, D') sig = Some (mD, rT0, body')" and
  lessD':  "G ⊢ X ⪯ Class D'" and
  suc_pc': "Suc pc' < length ins'"
  by (clarsimp simp add: wt_instr_def eff_def norm_eff_def) blast

from len len' ST0 ST0'
have "X = Class D" by simp
with lessD'
have "G ⊢ D ⪯C D'" by simp
moreover
```

```
    note wf meth'' methD'
    ultimately
    have "G ⊢ rT'' ≼ rT0" by (auto dest: subcls_widen_methd)
    with wf hd_stk rT''
    have hd_stk': "G,hp ⊢ (hd stk) ::≼ rT0" by (auto elim: conf_widen widen_trans)

    have frame'':
      "correct_frame G hp (ST'',LT'') maxl' ins' ?f'"
    proof -
      from wf hd_stk' len stk' less ST0
      have "approx_stk G hp (hd stk # drop (1+length pt) stk') ST''"
        by (auto simp add: map_eq_Cons sup_state_conv
                  dest!: approx_stk_append elim: conf_widen)
      moreover
      from wf loc' less
      have "approx_loc G hp loc' LT''" by (simp add: sup_state_conv) blast
      moreover
      note suc_pc' lloc'
      ultimately
      show ?thesis by (simp add: correct_frame_def)
    qed

    with state' frs' f meth hp_ok hd_stk phi_suc frames' meth' phi' class' alloc
    have ?thesis by (simp add: correct_state_def)
  }
  ultimately
  show ?thesis by (cases frs) blast+
qed

lemmas [simp] = map_append

lemma Goto_correct:
"[| wf_prog wt G;
  method (G,C) sig = Some (C,rT,maxs,maxl,ins,et);
  ins ! pc = Goto branch;
  wt_instr (ins!pc) G rT (phi C sig) maxs (length ins) et pc;
  Some state' = exec (G, None, hp, (stk,loc,C,sig,pc)#frs) ;
  G,phi ⊢JVM (None, hp, (stk,loc,C,sig,pc)#frs)√ |]
==> G,phi ⊢JVM state'√"
apply (clarsimp simp add: defs1)
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: defs1)
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: defs1)
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: defs1 map_eq_Cons)
apply (blast intro: conf_widen)
done

lemma Dup_x1_correct:
"[| wf_prog wt G;
  method (G,C) sig = Some (C,rT,maxs,maxl,ins,et);
  ins ! pc = Dup_x1;
  wt_instr (ins!pc) G rT (phi C sig) maxs (length ins) et pc;
  Some state' = exec (G, None, hp, (stk,loc,C,sig,pc)#frs) ;
  G,phi ⊢JVM (None, hp, (stk,loc,C,sig,pc)#frs)√ |]
==> G,phi ⊢JVM state'√"
apply (clarsimp simp add: defs1 map_eq_Cons)
apply (blast intro: conf_widen)
done

lemma Dup_x2_correct:
"[| wf_prog wt G;
  method (G,C) sig = Some (C,rT,maxs,maxl,ins,et);
  ins ! pc = Dup_x2;
  wt_instr (ins!pc) G rT (phi C sig) maxs (length ins) et pc;
  Some state' = exec (G, None, hp, (stk,loc,C,sig,pc)#frs) ;
  G,phi ⊢JVM (None, hp, (stk,loc,C,sig,pc)#frs)√ |]
==> G,phi ⊢JVM state'√"
apply (clarsimp simp add: defs1 map_eq_Cons)
apply (blast intro: conf_widen)
done
```

```
lemma Swap_correct:
"[| wf_prog wt G;
  method (G,C) sig = Some (C,rT,maxs,maxl,ins,et);
  ins ! pc = Swap;
  wt_instr (ins!pc) G rT (phi C sig) maxs (length ins) et pc;
  Some state' = exec (G, None, hp, (stk,loc,C,sig,pc)#frs) ;
  G,phi ⊢JVM (None, hp, (stk,loc,C,sig,pc)#frs)√ |]
==> G,phi ⊢JVM state'√"
apply (clarsimp simp add: defs1 map_eq_Cons)
apply (blast intro: conf_widen)
done

lemma IAdd_correct:
"[| wf_prog wt G;
  method (G,C) sig = Some (C,rT,maxs,maxl,ins,et);
  ins ! pc = IAdd;
  wt_instr (ins!pc) G rT (phi C sig) maxs (length ins) et pc;
  Some state' = exec (G, None, hp, (stk,loc,C,sig,pc)#frs) ;
  G,phi ⊢JVM (None, hp, (stk,loc,C,sig,pc)#frs)√ |]
==> G,phi ⊢JVM state'√"
apply (clarsimp simp add: defs1 map_eq_Cons approx_val_def conf_def)
apply blast
done

lemma Throw_correct:
"[| wf_prog wt G;
  method (G,C) sig = Some (C,rT,maxs,maxl,ins,et);
  ins ! pc = Throw;
  Some state' = exec (G, None, hp, (stk,loc,C,sig,pc)#frs) ;
  G,phi ⊢JVM (None, hp, (stk,loc,C,sig,pc)#frs)√;
  fst (exec_instr (ins!pc) G hp stk loc C sig pc frs) = None |]
==> G,phi ⊢JVM state'√"
  by simp
```

The next theorem collects the results of the sections above, i.e. exception handling and the execution step for each instruction. It states type safety for single step execution: in welltyped 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

35.4 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

end
```

36 Typing and Dataflow Analysis Framework

theory *Typing_Framework = Listn:*

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

types

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

constdefs

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

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

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

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

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

end

37 Kildall's Algorithm

theory *Kildall = Typing_Framework + While_Combinator + Product:*

syntax *"@lesubstep_type" :: "(nat × 's) list => 's ord => (nat × 's) list => bool"*
("(_ /<=|_| _)" [50, 0, 51] 50)
translations
"x <=|r| y" == "x <=[(Product.le (op =) r)] y"

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

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

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

primrec
"propa f [] ss w = (ss,w)"
"propa f (q'#qs) ss w = (let (q,t) = q';
u = t +_f ss!q;
w' = (if u = ss!q then w else insert q w)
in propa f qs (ss[q := u]) w')"

defs *iter_def:*
"iter f step ss w ==
while (%(ss,w). w ≠ {})
(%(ss,w). let p = SOME p. p ∈ w
in propa f (step p (ss!p)) ss (w-{p}))
(ss,w)"

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

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

consts *merges :: "'s binop => (nat × 's) list => 's list => 's list"*
primrec
"merges f [] ss = ss"
"merges f (p'#ps) ss = (let (p,s) = p' in merges f ps (ss[p := s +_f ss!p]))"

lemmas *[simp] = Let_def le_iff_plus_unchanged [symmetric]*

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

```
  "(x#xs) ++_f y = xs ++_f (x +_f y)"

lemma nth_merges:
  "!!ss. [| semilat (A, r, f); 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 sl: "semilat (A, r, f)"
  assume ss: "ss ∈ list n A"
  assume l:  "p < length ss"
  assume "?steptype (p'#ps')"
  then obtain a b where
    p': "p'=(a,b)" and ab: "a<n" "b∈A" and "?steptype ps'"
    by (cases p', auto)
  assume "⋀ss. semilat (A,r,f) ⟹ p < length ss ⟹ ss ∈ list n A ⟹ ?steptype ps' ⟹
?P ss ps'"
  hence IH: "⋀ss. ss ∈ list n A ⟹ p < length ss ⟹ ?P ss ps'" .

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


lemma 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 boundedD:
  "[| bounded step n; p < n; (q,t) : set (step p xs) |] ==> q < n"
  by (unfold bounded_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 length_merges [rule_format, simp]:
  "∀ss. size(merges f ps ss) = size ss"
  by (induct_tac ps, auto)


lemma merges_preserves_type_lemma:
  "[| semilat(A,r,f) |] ==>
     ∀xs. xs ∈ list n A --> (∀(p,x) ∈ set ps. p<n ∧ x∈A)
          --> merges f ps xs ∈ list n A"
  apply (frule semilatDclosedI)
  apply (unfold closed_def)
```

```
  apply (induct_tac ps)
   apply simp
  apply clarsimp
  done

lemma merges_preserves_type [simp]:
  "[| semilat(A,r,f); 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 merges_incr_lemma:
  "[| semilat(A,r,f) |] ==>
     ∀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 assumption
    apply simp
   apply simp
  apply (blast intro!: listE_set intro: closedD listE_length [THEN nth_in])
  done

lemma merges_incr:
  "[| semilat(A,r,f); 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 merges_same_conv [rule_format]:
  "[| semilat(A,r,f) |] ==>
     (∀xs. xs ∈ list n A --> (∀(p,x)∈set ps. p<size xs ∧ x∈A) -->
     (merges f ps xs = xs) = (∀(p,x)∈set ps. x <=_r xs!p))"
  apply (induct_tac ps)
   apply simp
  apply clarsimp
  apply (rename_tac p x ps xs)
  apply (rule iffI)
   apply (rule context_conjI)
    apply (subgoal_tac "xs[p := x +_f xs!p] <=[r] xs")
     apply (force dest!: le_listD simp add: nth_list_update)
    apply (erule subst, rule merges_incr)
        apply assumption
       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 (rotate_tac -2)
   apply (erule allE)
   apply (erule impE)
    apply assumption
   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 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 merges_pres_le_ub:
  "[| semilat(A,r,f); set ts <= A; set ss <= A;
     ∀(p,t)∈set ps. t <=_r ts!p ∧ t ∈ A ∧ p < size ts;
     ss <=[r] ts |]
  ==> merges f ps ss <=[r] ts"
proof -
  { fix A r f t ts ps
    have
    "!!qs. [| semilat(A,r,f); set ts <= A;
              ∀(p,t)∈set ps. t <=_r ts!p ∧ t ∈ A ∧ p < size ts |] ==>
    set qs <= set ps  -->
    (∀ss. set ss <= A --> ss <=[r] ts --> merges f qs ss <=[r] ts)"
    apply (induct_tac qs)
     apply simp
    apply (simp (no_asm_simp))
    apply clarify
    apply (rotate_tac -2)
    apply simp
    apply (erule allE, erule impE, erule_tac [2] mp)
     apply (drule bspec, assumption)
     apply (simp add: closedD)
    apply (drule bspec, assumption)
    apply (simp add: list_update_le_listI)
    done
  } note this [dest]

  case rule_context
  thus ?thesis by blast
qed




lemma decomp_propa:
  "!!ss w. (∀(q,t)∈set qs. q < size ss) ⟹
   propa f qs ss w =
   (merges f qs ss, {q. ∃t. (q,t)∈set qs ∧ t +_f ss!q ≠ ss!q} Un w)"
  apply (induct qs)
   apply simp
  apply (simp (no_asm))
  apply clarify
```

```
  apply simp
  apply (rule conjI)
   apply (simp add: nth_list_update)
   apply blast
  apply (simp add: nth_list_update)
  apply blast
  done



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

lemma ub2: "!!y. ⟦semilat (A, r, f); set x ⊆ A; y ∈ A⟧ ⟹ y <=_r x ++_f y"
proof (induct x)
  show "⋀y. semilat(A, r, f) ⟹ y <=_r [] ++_f y" by simp

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

  from sl have "order r" .. note order_trans [OF this, trans]

  from sl a y have "y <=_r a +_f y" by (rule semilat_ub2)
  also
  from sl a y have "a +_f y ∈ A" by (simp add: closedD)
  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 ub1: "⋀y. ⟦semilat (A, r, f); 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 sl: "semilat (A, r, f)"
  hence "order r" .. note order_trans [OF this, trans]
  assume "set (s#ls) ⊆ A"
  then obtain s: "s ∈ A" and ls: "set ls ⊆ A" by simp
  assume y: "y ∈ A"

  assume "⋀y. ⟦semilat (A, r, f); set ls ⊆ A; y ∈ A; x ∈ set ls⟧ ⟹ x <=_r ls ++_f y"
```

```
  hence IH: "⋀y. x ∈ set ls ⟹ y ∈ A ⟹ x <=_r ls ++_f y" .

  assume "x ∈ set (s#ls)"
  then obtain xls: "x = s ∨ x ∈ set ls" by simp
  moreover {
    assume xs: "x = s"
    from sl s y have "s <=_r s +_f y" by (rule semilat_ub1)
    also
    from sl s y have "s +_f y ∈ A" by (simp add: closedD)
    with sl ls have "(s +_f y) <=_r ls ++_f (s +_f y)" by (rule 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 sl s y
    have "s +_f y ∈ A" by (simp add: closedD)
    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 ub1':
  "⟦semilat (A, r, f); ∀(p,s) ∈ set S. s ∈ A; y ∈ A; (a,b) ∈ set S⟧
  ⟹ b <=_r map snd [(p', t')∈S. p' = a] ++_f y"
proof -
  let "b <=_r ?map ++_f y" = ?thesis

  assume "semilat (A, r, f)" "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 ub1)
qed



lemma plusplus_empty:
  "∀s'. (q, s') ∈ set S ⟶ s' +_f ss ! q = ss ! q ⟹
   (map snd [(p', t')∈ S. p' = q] ++_f ss ! q) = ss ! q"
apply (induct S)
apply auto
done


lemma stable_pres_lemma:
  "[| semilat (A,r,f); 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 assumption
      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 (frule nth_merges [of _ _ _ q _ _ "step p (ss!p)"])
   prefer 2 apply assumption
  apply simp
 apply clarify
 apply (rule conjI)
  apply (blast dest: boundedD)
 apply (erule pres_typeD)
    prefer 3 apply assumption
   apply simp
  apply simp
 apply (drule_tac P = "λx. (a, b) ∈ set (step q x)" in subst)
  apply assumption

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

apply simp
apply (erule allE, erule impE, assumption, erule impE, assumption)
apply (rule order_trans)
  apply simp
```

```
  defer
 apply (rule ub2)
    apply assumption
   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 lesub_step_type:
  "!!b x y. a <=|r| b ⟹ (x,y) ∈ set a ⟹ ∃y'. (x, y') ∈ set b ∧ y <=_r y'"
apply (induct a)
 apply simp
apply simp
apply (case_tac b)
 apply simp
apply simp
apply (erule disjE)
 apply clarify
 apply (simp add: lesub_def)
 apply blast
apply clarify
apply blast
done


lemma merges_bounded_lemma:
  "[| semilat (A,r,f); 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 assumption
    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_type, assumption)
  apply clarify
  apply (drule bspec, assumption)
```

```
  apply simp
  apply (blast intro: order_trans)
  done

lemma termination_lemma:
  "[| semilat(A,r,f); ss ∈ list n A; ∀(q,t)∈set qs. q<n ∧ t∈A; p∈w |] ==>
      ss <[r] merges f qs ss ∨
  merges f qs ss = ss ∧ {q. ∃t. (q,t)∈set qs ∧ t +_f ss!q ≠ ss!q} Un (w-{p}) < w"
  apply (unfold lesssub_def)
  apply (simp (no_asm_simp) add: merges_incr)
  apply (rule impI)
  apply (rule merges_same_conv [THEN iffD1, elim_format])
  apply assumption+
    defer
    apply (rule sym, assumption)
   defer apply simp
   apply (subgoal_tac "∀q t. ¬((q, t) ∈ set qs ∧ t +_f ss ! q ≠ ss ! q)")
   apply (blast intro!: psubsetI elim: equalityE)
   apply clarsimp
   apply (drule bspec, assumption)
   apply (drule bspec, assumption)
   apply clarsimp
  done

lemma iter_properties[rule_format]:
  "⟦ semilat(A,r,f); acc r ; pres_type step n A; mono r step n A;
     bounded step n; ∀p∈w0. p < n; ss0 ∈ list n A;
     ∀p<n. p ∉ w0 ⟶ stable r step ss0 p ⟧ ⟹
   iter f step ss0 w0 = (ss',w')
   ⟶
   ss' ∈ list n A ∧ stables r step ss' ∧ ss0 <=[r] ss' ∧
   (∀ts∈list n A. ss0 <=[r] ts ∧ stables r step ts ⟶ ss' <=[r] ts)"
apply (unfold iter_def stables_def)
apply (rule_tac P = "%(ss,w).
 ss ∈ list n A ∧ (∀p<n. p ∉ w ⟶ stable r step ss p) ∧ ss0 <=[r] ss ∧
 (∀ts∈list n A. ss0 <=[r] ts ∧ stables r step ts ⟶ ss <=[r] ts) ∧
 (∀p∈w. p < n)" and
 r = "{(ss',ss) . ss <[r] ss'} <*lex*> finite_psubset"
       in while_rule)

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

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

```
 apply blast
apply (subst decomp_propa)
 apply blast
apply simp
apply (rule conjI)
 apply (erule merges_preserves_type)
 apply blast
 apply clarify
 apply (rule conjI)
  apply(clarsimp, blast dest!: boundedD)
 apply (erule pres_typeD)
  prefer 3
  apply assumption
  apply (erule listE_nth_in)
  apply blast
 apply blast
apply (rule conjI)
 apply clarify
 apply (blast intro!: stable_pres_lemma)
apply (rule conjI)
 apply (blast intro!: merges_incr intro: le_list_trans)
apply (rule conjI)
 apply clarsimp
 apply (blast intro!: merges_bounded_lemma)
apply (blast dest!: boundedD)
```

— Postcondition holds upon termination:

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

— Well-foundedness of the termination relation:

```
apply (rule wf_lex_prod)
 apply (drule (1) semilatDorderI [THEN acc_le_listI])
 apply (simp only: acc_def lesssub_def)
apply (rule wf_finite_psubset)
```

— Loop decreases along termination relation:

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

```
apply (rule termination_lemma)
apply assumption+
defer
apply assumption
apply clarsimp
apply (blast dest!: boundedD)
done


lemma kildall_properties:
  "⟦ semilat(A,r,f); acc r; pres_type step n A; mono r step n A;
     bounded step n; ss0 ∈ list n A ⟧ ⟹
  kildall r f step ss0 ∈ list n A ∧
  stables r step (kildall r f step ss0) ∧
  ss0 <=[r] kildall r f step ss0 ∧
  (∀ts∈list n A. ss0 <=[r] ts ∧ stables r step ts ⟶
                 kildall r f step ss0 <=[r] ts)"
apply (unfold kildall_def)
apply(case_tac "iter f step ss0 (unstables r step ss0)")
apply(simp)
apply (rule iter_properties)
apply (simp_all add: unstables_def stable_def)
done

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

end
```

38 Static and Dynamic Welltyping

theory *Typing_Framework_err = Typing_Framework:*

constdefs

```
dynamic_wt :: "'s ord => 's err step_type => 's err list => bool"
"dynamic_wt r step ts == wt_step (Err.le r) Err step ts"

static_wt :: "'s ord => (nat => 's => bool) => 's step_type => 's list => bool"
"static_wt r app step ts ==
  ∀p < size ts. app p (ts!p) ∧ (∀(q,t) ∈ set (step p (ts!p)). t <=_r ts!q)"

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

map_err :: "('a × 'b) list ⇒ ('a × 'b err) list"
"map_err == map_snd (λy. Err)"

err_step :: "(nat ⇒ 's ⇒ bool) ⇒ 's step_type ⇒ 's err step_type"
"err_step app step p t == case t of Err ⇒ map_err (step p arbitrary) | OK t' ⇒
  if app p t' then map_snd OK (step p t') else map_err (step p t')"

non_empty :: "'s step_type => bool"
"non_empty step == ∀p t. step p t ≠ []"
```

lemmas *err_step_defs = err_step_def map_snd_def map_err_def*

lemma *non_emptyD:*
  "non_empty step ==> ∃q t'. (q,t') ∈ set(step p t)"
proof *(unfold non_empty_def)*
  assume *"∀p t. step p t ≠ []"*
  hence *"step p t ≠ []"* **by** *blast*
  then obtain *x xs* **where** *"step p t = x#xs"*
    by *(auto simp add: neq_Nil_conv)*
  hence *"x ∈ set(step p t)"* **by** *simp*
  thus *?thesis* **by** *(cases x) blast*
qed

lemma *dynamic_imp_static:*
  "[| bounded step (size ts); non_empty step;
  dynamic_wt r (err_step app step) ts |]
  ==> static_wt r app step (map ok_val ts)"
proof *(unfold static_wt_def, intro strip, rule conjI)*

  assume *b: "bounded step (size ts)"*
  assume *n: "non_empty step"*
  assume *wt: "dynamic_wt r (err_step app step) ts"*

  fix *p*
  assume *"p < length (map ok_val ts)"*
  hence *lp: "p < length ts"* **by** *simp*

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

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

    from n
    obtain q t where q: "(q,t) ∈ set(step p s)"
      by (blast dest: non_emptyD)

    from lp b q
    have lq: "q < length ts" by (unfold bounded_def) blast
    hence "ts!q ≠ Err" ..
    then obtain s' where OKq: "ts ! q = OK s'" by (auto iff: not_Err_eq)

    with OKp less q have "app p s"
      by (auto simp add: err_step_defs split: err.split_asm split_if_asm)

    with lp OKp show ?thesis by simp
  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 app step p (ts!p)). t <=_(Err.le r) ts!q"
      by (unfold dynamic_wt_def wt_step_def stable_def)
         (auto iff: not_Err_eq)

    from lp b q
    have lq: "q < length ts" by (unfold bounded_def) blast
    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 static_imp_dynamic:
  "[| static_wt r app step ts; bounded step (size ts) |]
  ==> dynamic_wt r (err_step app step) (map OK ts)"
proof (unfold dynamic_wt_def wt_step_def, intro strip, rule conjI)
  assume bounded: "bounded step (size ts)"
  assume static:  "static_wt r app step ts"
  fix p assume "p < length (map OK ts)"
  hence p: "p < length ts" by simp
  thus "map OK ts ! p ≠ Err" by simp
  { fix q t
    assume q: "(q,t) ∈ set (err_step app step p (map OK ts ! p))"
    with p static obtain
```

```
      "app p (ts ! p)" "∀(q,t) ∈ set (step p (ts!p)). t <=_r ts!q"
      by (unfold static_wt_def) blast
    moreover
    from q p bounded have "q < size ts"
      by (clarsimp simp add: bounded_def err_step_defs
          split: err.splits split_if_asm) blast+
    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 app step) (map OK ts) p"
    by (unfold stable_def) blast
qed

end



theory Kildall_Lift = Kildall + Typing_Framework_err:

constdefs
 app_mono :: "'s ord ⇒ (nat ⇒ 's ⇒ bool) ⇒ nat ⇒ 's set ⇒ bool"
"app_mono r app n A ==
 ∀s p t. s ∈ A ∧ p < n ∧ s <=_r t ⟶ app p t ⟶ app p s"

 succs_stable :: "'s ord ⇒ 's step_type ⇒ bool"
"succs_stable r step == ∀p t t'. map fst (step p t') = map fst (step p t)"

lemma succs_stableD:
  "succs_stable r step ⟹ map fst (step p t) = map fst (step p t')"
  by (unfold succs_stable_def) blast

lemma eqsub_def [simp]: "a <=_(op =) b = (a = b)" by (simp add: lesub_def)

lemma list_le_eq [simp]: "⋀b. a <=[op =] b = (a = b)"
apply (induct a)
 apply simp
 apply rule
  apply simp
 apply simp
apply (case_tac b)
 apply simp
apply simp
done

lemma map_err: "map_err a = zip (map fst a) (map (λx. Err) (map snd a))"
apply (induct a)
apply (auto simp add: map_err_def map_snd_def)
done

lemma map_snd: "map_snd f a = zip (map fst a) (map f (map snd a))"
apply (induct a)
apply (auto simp add: map_snd_def)
done
```

```
lemma zipI:
  "⋀b c d. a <=[r1] c ⟹ b <=[r2] d ⟹ zip a b <=[Product.le r1 r2] zip c d"
apply (induct a)
 apply simp
apply (case_tac c)
 apply simp
apply (case_tac b)
 apply simp
apply (case_tac d)
 apply simp
apply simp
done

lemma step_type_ord:
  "⋀b. a <=|r| b ⟹ map snd a <=[r] map snd b ∧ map fst a = map fst b"
apply (induct a)
 apply simp
apply (case_tac b)
 apply simp
apply simp
apply (auto simp add: Product.le_def lesub_def)
done

lemma map_OK_Err:
  "⋀y. length y = length x ⟹ map OK x <=[Err.le r] map (λx. Err) y"
apply (induct x)
 apply simp
apply simp
apply (case_tac y)
apply auto
done

lemma map_Err_Err:
  "⋀b. length a = length b ⟹ map (λx. Err) a <=[Err.le r] map (λx. Err) b"
apply (induct a)
 apply simp
apply (case_tac b)
apply auto
done

lemma succs_stable_length:
  "succs_stable r step ⟹ length (step p t) = length (step p t')"
proof -
  assume "succs_stable r step"
  hence "map fst (step p t) = map fst (step p t')" by (rule succs_stableD)
  hence "length (map fst (step p t)) = length (map fst (step p t'))" by simp
  thus ?thesis by simp
qed

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 mono_lift:
  "order r ⟹ succs_stable r step ⟹ app_mono r app n A ⟹
  ∀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 app step) n (err A)"
apply (unfold app_mono_def mono_def err_step_def)
apply clarify
apply (case_tac s)
 apply simp
 apply (rule le_list_refl)
 apply simp
apply simp
apply (subgoal_tac "map fst (step p arbitrary) = map fst (step p a)")
 prefer 2
 apply (erule succs_stableD)
apply (case_tac t)
 apply simp
 apply (rule conjI)
  apply clarify
  apply (simp add: map_err map_snd)
  apply (rule zipI)
   apply simp
  apply (rule map_OK_Err)
  apply (subgoal_tac "length (step p arbitrary) = length (step p a)")
   prefer 2
   apply (erule succs_stable_length)
  apply simp
 apply clarify
 apply (simp add: map_err)
 apply (rule zipI)
  apply simp
 apply (rule map_Err_Err)
 apply simp
 apply (erule succs_stable_length)
apply simp
apply (elim allE)
apply (erule impE, blast)+
apply (rule conjI)
 apply clarify
 apply (simp add: map_snd)
 apply (rule zipI)
  apply simp
  apply (erule succs_stableD)
 apply (simp add: step_type_ord)
apply clarify
apply (rule conjI)
 apply clarify
 apply (simp add: map_snd map_err)
 apply (rule zipI)
  apply simp
  apply (erule succs_stableD)
 apply (rule map_OK_Err)
 apply (simp add: succs_stable_length)
apply clarify
apply (simp add: map_err)
apply (rule zipI)
 apply simp
 apply (erule succs_stableD)
apply (rule map_Err_Err)
apply (simp add: succs_stable_length)
```

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_lift:*
"bounded (err_step app step) n = bounded step n"
apply *(unfold bounded_def err_step_def)*
apply *rule*
apply *clarify*
apply *(erule allE, erule impE, assumption)*
apply *(erule_tac x = "OK s" in allE)*
apply *simp*
apply *(case_tac "app p s")*
apply *(simp add: map_snd_def)*
apply *(drule bspec, assumption)*
apply *simp*
apply *(simp add: map_err_def map_snd_def)*
apply *(drule bspec, assumption)*
apply *simp*
apply *clarify*
apply *(case_tac s)*
apply *simp*
apply *(simp add: map_err_def)*
apply *(blast dest: in_map_sndD)*
apply *(simp split: split_if_asm)*
apply *(blast dest: in_map_sndD)*
apply *(simp add: map_err_def)*
apply *(blast dest: in_map_sndD)*
done

lemma *set_zipD: "⋀y. (a,b) ∈ set (zip x y) ⟹ (a ∈ set x ∧ b ∈ set y)"*
apply *(induct x)*
apply *simp*
apply *(case_tac y)*
apply *simp*
apply *simp*
apply *blast*
done

lemma *pres_type_lift:*
"∀s∈A. ∀p. p < n ⟶ app p s ⟶ (∀(q, s')∈set (step p s). s' ∈ A)
⟹ pres_type (err_step 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 add: map_err)*
apply *(drule set_zipD)*
apply *clarify*
apply *simp*
apply *blast*
apply *(simp add: map_err split: split_if_asm)*
apply *(simp add: map_snd_def)*
apply *fastsimp*
apply *(drule set_zipD)*

apply *simp*
apply *blast*
done

end

39 Monotonicity of eff and app

```
theory EffectMono = Effect:


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


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

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

    show "(∃t. (a # list) ! n = OK t) ∧ G ⊢(a # list) ! n <=o OK t"
    proof (cases n)
      case 0
      with * show ?thesis by (simp add: sup_ty_opt_OK)
    next
      case Suc
      with Cons *
      show ?thesis by (simp add: sup_loc_def Listn.le_def lesub_def)
    qed
  qed
qed


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

  fix l ls assume Cons: "?P ls"

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


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

```
  fix l ls assume Cons: "?P ls"

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

    show "∃a b. l # ls = a @ b ∧ length a = n"
    proof (cases n)
      assume "n=0" thus ?thesis by simp
    next
      fix n' assume s: "n = Suc n'"
      with l have  "n' ≤ length ls" by simp
      hence "∃a b. ls = a @ b ∧ length a = n'" by (rule Cons [rule_format])
      then obtain a b where "ls = a @ b" "length a = n'" by rules
      with s have "l # ls = (l#a) @ b ∧ length (l#a) = n" by simp
      thus ?thesis by blast
    qed
  qed
qed

lemma rev_append_cons:
"n < length x ==> ∃a b c. x = (rev a) @ b # c ∧ length a = n"
proof -
  assume n: "n < length x"
  hence "n ≤ length x" by simp
  hence "∃a b. x = a @ b ∧ length a = n" by (rule append_length_n)
  then obtain r d where x: "x = r@d" "length r = n" by rules
  with n have "∃b c. d = b#c" by (simp add: neq_Nil_conv)
  then obtain b c where "d = b#c" by rules
  with x have "x = (rev (rev r)) @ b # c ∧ length (rev r) = n" by simp
  thus ?thesis by blast
qed

lemma sup_loc_length_map:
  "G ⊢ map f a <=l map g b ⟹ length a = length b"
proof -
  assume "G ⊢ map f a <=l map g b"
  hence "length (map f a) = length (map g b)" by (rule sup_loc_length)
  thus ?thesis by simp
qed

lemmas [iff] = not_Err_eq

lemma app_mono:
"[|G ⊢ s <=' s'; app i G m rT pc et s'|] ==> app i G m rT pc et s"
proof -

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

    note [simp] = sup_loc_length sup_loc_length_map

    have "app i G m rT pc et (Some s2)"
    proof (cases (open) i)
      case Load
```

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

  with G Load app show ?thesis
    by (cases s2) (auto simp add: sup_state_conv dest: sup_loc_some)
next
  case Store
  with G app show ?thesis
    by (cases s2, auto simp add: map_eq_Cons sup_loc_Cons2 sup_state_conv)
next
  case LitPush
  with G app show ?thesis by (cases s2, auto simp add: sup_state_conv)
next
  case New
  with G app show ?thesis by (cases s2, auto simp add: sup_state_conv)
next
  case Getfield
  with app G show ?thesis
    by (cases s2) (clarsimp simp add: sup_state_Cons2, rule widen_trans)
next
  case Putfield

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

```
    case Dup
    with app G show ?thesis
      by (cases s2, clarsimp simp add: sup_state_Cons2,
          auto dest: sup_state_length)
  next
    case Dup_x1
    with app G show ?thesis
      by (cases s2, clarsimp simp add: sup_state_Cons2,
          auto dest: sup_state_length)
  next
    case Dup_x2
    with app G show ?thesis
      by (cases s2, clarsimp simp add: sup_state_Cons2,
          auto dest: sup_state_length)
  next
    case Swap
    with app G show ?thesis
      by (cases s2, clarsimp simp add: sup_state_Cons2)
  next
    case IAdd
    with app G show ?thesis
      by (cases s2, auto simp add: sup_state_Cons2 PrimT_PrimT)
  next
    case Goto
    with app show ?thesis by simp
  next
    case Ifcmpeq
    with app G show ?thesis
      by (cases s2, auto simp add: sup_state_Cons2 PrimT_PrimT widen_RefT2)
  next
    case Invoke

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

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

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

```
      from this s1 s2 G
      obtain
        G': "G ⊢ (apTs',LT') <=s (apTs,LT)" and
        X : "G ⊢  X' ⪯ X" and "G ⊢ (ST',LT') <=s (ST,LT)"
        by (simp add: sup_state_rev_fst sup_state_append_fst sup_state_Cons1)

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

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

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

      from Invoke s2 l' w' C' m c x
      show ?thesis
        by (simp del: split_paired_Ex) blast
    next
      case Throw
      with app G show ?thesis
        by (cases s2, clarsimp simp add: sup_state_Cons2 widen_RefT2)
    qed
  } note this [simp]

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

lemmas [simp del] = split_paired_Ex

lemma eff'_mono:
"[| app i G m rT pc et (Some s2); G ⊢ s1 <=s s2 |] ==>
  G ⊢ eff' (i,G,s1) <=s eff' (i,G,s2)"
proof (cases s1, cases s2)
  fix a1 b1 a2 b2
  assume s: "s1 = (a1,b1)" "s2 = (a2,b2)"
  assume app2: "app i G m rT pc et (Some s2)"
  assume G: "G ⊢ s1 <=s s2"

  note [simp] = eff_def

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

  show ?thesis
  proof (cases (open) i)
    case Load
```

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

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

  from G s
  have "G ⊢ b1 <=l b2" by (simp add: sup_state_conv)

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

  with Load G y y' s app1 app2
  show ?thesis by (clarsimp simp add: sup_state_conv)
next
  case Store
  with G s app1 app2
  show ?thesis
    by (clarsimp simp add: sup_state_conv sup_loc_update)
next
  case LitPush
  with G s app1 app2
  show ?thesis
    by (clarsimp simp add: sup_state_Cons1)
next
  case New
  with G s app1 app2
  show ?thesis
    by (clarsimp simp add: sup_state_Cons1)
next
  case Getfield
  with G s app1 app2
  show ?thesis
    by (clarsimp simp add: sup_state_Cons1)
next
  case Putfield
  with G s app1 app2
  show ?thesis
    by (clarsimp simp add: sup_state_Cons1)
next
  case Checkcast
  with G s app1 app2
  show ?thesis
    by (clarsimp simp add: sup_state_Cons1)
next
  case Invoke

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

  from Invoke s app2
  obtain a' X' ST' where
    s2: "s2 = (a' @ X' # ST', b2)" and
```

```
      l': "length a' = length list"
      by (simp, elim exE conjE, simp)

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

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

    moreover

    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
```

40 Kildall for the JVM

theory *JVM = Kildall_Lift + JVMType + Opt + Product + Typing_Framework_err + EffectMono + BVSpec:*

constdefs

```
exec :: "jvm_prog ⇒ nat ⇒ ty ⇒ exception_table ⇒ instr list ⇒ state step_type"
"exec G maxs rT et bs ==
err_step (λpc. app (bs!pc) G maxs rT pc et) (λpc. eff (bs!pc) G pc et)"

kiljvm :: "jvm_prog => nat => nat => ty => exception_table =>
           instr list => state list => state list"
"kiljvm G maxs maxr rT et bs ==
kildall (JVMType.le G maxs maxr) (JVMType.sup G maxs maxr) (exec G maxs rT et bs)"

wt_kil :: "jvm_prog ⇒ cname ⇒ ty list ⇒ ty ⇒ nat ⇒ nat ⇒
           exception_table ⇒ instr list ⇒ bool"
"wt_kil G C pTs rT mxs mxl et ins ==
 bounded (exec G mxs rT et ins) (size ins) ∧ 0 < size ins ∧
 (let first  = Some ([],(OK (Class C))#((map OK pTs))@(replicate mxl Err));
      start  = OK first#(replicate (size ins - 1) (OK None));
      result = kiljvm G mxs (1+size pTs+mxl) rT et ins start
  in ∀n < size ins. result!n ≠ Err)"

wt_jvm_prog_kildall :: "jvm_prog => bool"
"wt_jvm_prog_kildall G ==
wf_prog (λG C (sig,rT,(maxs,maxl,b,et)). wt_kil G C (snd sig) rT maxs maxl et b) G"
```

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*

lemma *non_empty_succs: "succs i pc ≠ []"*
by *(cases i) auto*

lemma *non_empty:*
"non_empty (λpc. eff (bs!pc) G pc et)"
by *(simp add: non_empty_def eff_def non_empty_succs)*

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

lemma *listn_appendE [elim!]:*
"a@b ∈ list n A ⟹ (⋀n1 n2. n=n1+n2 ⟹ a ∈ list n1 A ⟹ b ∈ list n2 A ⟹ P) ⟹ P"

proof -
have *"⋀n. a@b ∈ list n A ⟹ ∃n1 n2. n=n1+n2 ∧ a ∈ list n1 A ∧ b ∈ list n2 A"*
(**is** *"⋀n. ?list a n ⟹ ∃n1 n2. ?P a n n1 n2"*)

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


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
apply (rule_tac x=1 in exI)
apply fastsimp

apply clarsimp
apply (erule disjE)
 apply fastsimp
apply simp
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 (drule listE_length)+
apply fastsimp

apply clarsimp
apply (rule_tac x="Suc (Suc (Suc (length ST)))" in exI)
apply simp
apply (drule listE_length)+
apply fastsimp

apply clarsimp
apply (rule_tac x="Suc (Suc (Suc (Suc (length ST))))" in exI)
apply simp
apply (drule listE_length)+
apply fastsimp

apply fastsimp
apply fastsimp
apply fastsimp
apply fastsimp

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

```
  apply fastsimp

  apply (erule disjE)
   apply (clarsimp simp add: Un_subset_iff)
   apply (drule method_wf_mdecl, assumption+)
   apply (clarsimp simp add: wf_mdecl_def wf_mhead_def)
   apply fastsimp
  apply clarsimp
  apply (rule_tac x=1 in exI)
  apply fastsimp
  done

lemmas [iff] = not_None_eq

lemma map_fst_eq:
  "map fst (map (λz. (f z, x z)) a) = map fst (map (λz. (f z, y z)) a)"
  by (induct a) auto

lemma succs_stable_eff:
  "succs_stable (sup_state_opt G) (λpc. eff (bs!pc) G pc et)"
  apply (unfold succs_stable_def eff_def xcpt_eff_def)
  apply (simp add: map_fst_eq)
  done

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)

constdefs
  opt_states :: "'c prog ⇒ nat ⇒ nat ⇒ (ty list × ty err list) option set"
  "opt_states G maxs maxr ≡ opt (⋃{list n (types G) |n. n ≤ maxs} × list maxr (err (types
G)))"

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 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 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 le_list_appendI)
   apply (simp add: norm_eff_def)
   apply (rule le_listI)
    apply simp
   apply simp
   apply (simp add: lesub_def)
   apply (case_tac s)
    apply simp
   apply (simp del: split_paired_All split_paired_Ex)
   apply (elim exE conjE)
   apply simp
   apply (drule eff'_mono, assumption)
   apply assumption
  apply (simp add: xcpt_eff_def)
  apply (rule le_listI)
    apply simp
  apply simp
  apply (simp add: lesub_def)
  apply (case_tac s)
   apply simp
  apply simp
  apply (case_tac t)
   apply simp
  apply (clarsimp simp add: sup_state_conv)
  done

lemma order_sup_state_opt:
  "wf_prog wf_mb G ⟹ order (sup_state_opt G)"
  by (unfold sup_state_opt_unfold) (blast dest: acyclic_subcls1 order_widen)

theorem exec_mono:
  "wf_prog wf_mb G ==>
  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 succs_stable_eff)
   apply (rule app_mono)
  apply clarify
  apply (rule eff_mono)
  apply assumption+
  done

theorem semilat_JVM_slI:
  "wf_prog wf_mb G ==> semilat (JVMType.sl G maxs maxr)"
  apply (unfold JVMType.sl_def stk_esl_def reg_sl_def)
  apply (rule semilat_opt)
```

```
  apply (rule err_semilat_Product_esl)
  apply (rule err_semilat_upto_esl)
  apply (rule err_semilat_JType_esl, assumption+)
  apply (rule err_semilat_eslI)
  apply (rule semilat_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)


theorem is_bcv_kiljvm:
  "[| wf_prog wf_mb G; bounded (exec G maxs rT et bs) (size bs) |] ==>
      is_bcv (JVMType.le G maxs maxr) Err (exec G maxs rT et bs)
             (size bs) (states G maxs maxr) (kiljvm G maxs maxr rT et bs)"
  apply (unfold kiljvm_def sl_triple_conv)
  apply (rule is_bcv_kildall)
       apply (simp (no_asm) add: sl_triple_conv [symmetric])
       apply (force intro!: semilat_JVM_slI dest: wf_acyclic simp add: symmetric sl_triple_conv)
      apply (simp (no_asm) add: JVM_le_unfold)
      apply (blast intro!: order_widen wf_converse_subcls1_impl_acc_subtype
                   dest: wf_subcls1 wf_acyclic)
     apply (simp add: JVM_le_unfold)
    apply (erule exec_pres_type)
    apply assumption
  apply (erule exec_mono)
  done


theorem wt_kil_correct:
  "[| wt_kil G C pTs rT maxs mxl et bs; wf_prog wf_mb G;
      is_class G C; ∀x ∈ set pTs. is_type G x |]
  ==> ∃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))"

  assume wf:      "wf_prog wf_mb G"
  assume isclass: "is_class G C"
  assume istype:  "∀x ∈ set pTs. is_type G x"

  assume "wt_kil G C pTs rT maxs mxl et bs"
  then obtain maxr r where
    bounded: "bounded (exec G maxs rT et bs) (size bs)" 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 wf bounded
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)

{ fix l x have "set (replicate l x) ⊆ {x}" by (cases "0 < l") simp+
} note subset_replicate = this
from istype have "set pTs ⊆ types G" by auto
hence "OK ` set pTs ⊆ err (types G)" by auto
with instrs maxr isclass
have "?start ∈ list (length bs) (states G maxs maxr)"
  apply (unfold list_def JVM_states_unfold)
  apply simp
  apply (rule conjI)
   apply (simp add: Un_subset_iff)
   apply (rule_tac B = "{Err}" in subset_trans)
    apply (simp add: subset_replicate)
   apply simp
  apply (rule_tac B = "{OK None}" in subset_trans)
   apply (simp add: subset_replicate)
  apply simp
  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
  l: "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 dynamic:
  "dynamic_wt (sup_state_opt G) (exec G maxs rT et bs) phi'"
  by (simp add: dynamic_wt_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 l 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 l bounded
have bounded': "bounded (λpc. eff (bs!pc) G pc et) (length phi')"
  by (simp add: exec_def bounded_lift)
with dynamic
have "static_wt (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: dynamic_imp_static simp add: exec_def non_empty)
  with instrs l le bounded'
  have "wt_method G C pTs rT maxs mxl bs et (map ok_val phi')"
    apply (unfold wt_method_def static_wt_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 (unfold bounded_def)
    apply blast
    done

  thus ?thesis by blast
qed


theorem wt_kil_complete:
  "[| wt_method G C pTs rT maxs mxl bs et phi; wf_prog wf_mb G;
      length phi = length bs; is_class G C; ∀x ∈ set pTs. is_type G x;
      map OK phi ∈ list (length bs) (states G maxs (1+size pTs+mxl)) |]
  ==> wt_kil G C pTs rT maxs mxl et bs"
proof -
  assume wf: "wf_prog wf_mb G"
  assume isclass: "is_class G C"
  assume istype: "∀x ∈ set pTs. is_type G x"
  assume length: "length phi = length bs"
  assume istype_phi: "map OK phi ∈ list (length bs) (states G maxs (1+size pTs+mxl))"

  assume "wt_method G C pTs rT maxs mxl bs et phi"
  then obtain
    instrs:   "0 < length bs" 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

  let ?eff  = "λpc. eff (bs!pc) G pc et"
  let ?app  = "λpc. app (bs!pc) G maxs rT pc et"

  have bounded_eff: "bounded ?eff (size bs)"
  proof (unfold bounded_def, clarify)
    fix pc pc' s s' assume "pc < length bs"
    with wt_ins have "wt_instr (bs!pc) G rT phi maxs (length bs) et pc" by fast
    then obtain "∀(pc',s') ∈ set (?eff pc (phi!pc)). pc' < length bs"
      by (unfold wt_instr_def) fast
    hence "∀pc' ∈ set (map fst (?eff pc (phi!pc))). pc' < length bs" by auto
    also
```

```
  note succs_stable_eff
  hence "map fst (?eff pc (phi!pc)) = map fst (?eff pc s)"
    by (rule succs_stableD)
  finally have "∀(pc',s') ∈ set (?eff pc s). pc' < length bs" by auto
  moreover assume "(pc',s') ∈ set (?eff pc s)"
  ultimately show "pc' < length bs" by blast
qed
hence bounded_exec: "bounded (exec G maxs rT et bs) (size bs)"
  by (simp add: exec_def bounded_lift)

from wt_ins
have "static_wt (sup_state_opt G) ?app ?eff phi"
  apply (unfold static_wt_def wt_instr_def lesub_def)
  apply (simp (no_asm) only: length)
  apply blast
  done

with bounded_eff
have "dynamic_wt (sup_state_opt G) (err_step ?app ?eff) (map OK phi)"
  by - (erule static_imp_dynamic, simp add: length)
hence dynamic:
  "dynamic_wt (sup_state_opt G) (exec G maxs rT et bs) (map OK phi)"
  by (unfold exec_def)

let ?maxr = "1+size pTs+mxl"
from wf bounded_exec
have is_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)

let ?start = "OK (Some ([],(OK (Class C))#((map OK pTs))@(replicate mxl Err)))
              #(replicate (size bs - 1) (OK None))"

{ fix l x have "set (replicate l x) ⊆ {x}" by (cases "0 < l") simp+
} note subset_replicate = this

from istype have "set pTs ⊆ types G" by auto
hence "OK ` set pTs ⊆ err (types G)" by auto
with instrs isclass have start:
  "?start ∈ list (length bs) (states G maxs ?maxr)"
  apply (unfold list_def JVM_states_unfold)
  apply simp
  apply (rule conjI)
   apply (simp add: Un_subset_iff)
   apply (rule_tac B = "{Err}" in subset_trans)
    apply (simp add: subset_replicate)
   apply simp
  apply (rule_tac B = "{OK None}" in subset_trans)
   apply (simp add: subset_replicate)
  apply simp
  done
```

```
  let ?phi = "map OK phi"
  have less_phi: "?start <=[JVMType.le G maxs ?maxr] ?phi"
  proof -
    from length 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 length
      have "0 < length phi" by simp
      ultimately
      have "JVMType.le G maxs ?maxr (?start!0) (?phi!0)"
        by (simp add: JVM_le_Err_conv Err.le_def lesub_def)
      moreover
      { fix n'
        have "JVMType.le G maxs ?maxr (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 ?maxr (?start!n) (?phi!n)"
          by simp
      }
      ultimately
      have "n < length ?start ==> (?start!n) <=_(JVMType.le G maxs ?maxr) (?phi!n)"
        by (unfold lesub_def) (cases n, blast+)
    }
    ultimately show ?thesis by (rule le_listI)
  qed

  from dynamic
  have "wt_step (JVMType.le G maxs ?maxr) Err (exec G maxs rT et bs) ?phi"
    by (simp add: dynamic_wt_def JVM_le_Err_conv)
  with start istype_phi less_phi is_bcv
  have "∀p. p < length bs ⟶ kiljvm G maxs ?maxr rT et bs ?start ! p ≠ Err"
    by (unfold is_bcv_def) auto
  with bounded_exec instrs
  show "wt_kil G C pTs rT maxs mxl et bs" by (unfold wt_kil_def) simp
qed
```

The above theorem `wt_kil_complete` is all nice'n shiny except for one assumption: `map OK phi ∈ list (length bs) (states G maxs (1 + size pTs + mxl))` It does not hold for all `phi` that satisfy `wt_method`.

The assumption states mainly that all entries in `phi` are legal types in the program context, that the stack size is bounded by `maxs`, and that the register sizes are exactly `1 + size pTs + mxl`. The BV specification, i.e. `wt_method`, only gives us this property for *reachable* code. For unreachable code, e.g. unused registers may contain arbitrary garbage. Even the stack and register sizes can be different from the rest of the program (as long as they are consistent inside each chunk of unreachable code).

All is not lost, though: for each *phi* that satisfies *wt_method* there is a *phi'* that also satisfies *wt_method* and that additionally satisfies our assumption. The construction is quite easy: the entries for reachable code are the same in *phi* and *phi'*, the entries for unreachable code are all *None* in *phi'* (as it would be produced by Kildall's algorithm).

Although this is proved easily by comment, it requires some more overhead (i.e. talking about reachable instructions) if you try it the hard way. Thus it is missing here for the time being.

The other direction (*wt_kil_correct*) can be lifted to programs without problems:

```
lemma is_type_pTs:
  "[| wf_prog wf_mb G; (C,S,fs,mdecls) ∈ set G; (sig,rT,code) ∈ set mdecls;
      t ∈ set (snd sig) |]
  ==> is_type G t"
proof -
  assume "wf_prog wf_mb G"
         "(C,S,fs,mdecls) ∈ set G"
         "(sig,rT,code) ∈ set mdecls"
  hence "wf_mdecl wf_mb G C (sig,rT,code)"
    by (unfold wf_prog_def wf_cdecl_def) auto
  hence "∀t ∈ set (snd sig). is_type G t"
    by (unfold wf_mdecl_def wf_mhead_def) auto
  moreover
  assume "t ∈ set (snd sig)"
  ultimately
  show ?thesis by blast
qed


theorem jvm_kildall_correct:
  "wt_jvm_prog_kildall G ==> ∃Phi. wt_jvm_prog G Phi"
proof -
  assume wtk: "wt_jvm_prog_kildall G"

  then obtain wf_mb where
    wf: "wf_prog wf_mb G"
    by (auto simp add: wt_jvm_prog_kildall_def)

  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"

  { fix C S fs mdecls sig rT code
    assume "(C,S,fs,mdecls) ∈ set G" "(sig,rT,code) ∈ set mdecls"
    with wf
    have "method (G,C) sig = Some (C,rT,code) ∧ is_class G C ∧ (∀t ∈ set (snd sig). is_type
G t)"
      by (simp add: methd is_type_pTs)
  } note this [simp]

  from wtk
  have "wt_jvm_prog G ?Phi"
    apply (unfold wt_jvm_prog_def wt_jvm_prog_kildall_def wf_prog_def wf_cdecl_def)
    apply clarsimp
    apply (drule bspec, assumption)
    apply (unfold wf_mdecl_def)
    apply clarsimp
    apply (drule bspec, assumption)
    apply clarsimp
    apply (drule wt_kil_correct [OF _ wf])
    apply (auto intro: someI)
```

done

thus *?thesis* **by** *blast*
qed

end

41 The Lightweight Bytecode Verifier

theory *LBVSpec = Effect + Kildall:*

The Lightweight Bytecode Verifier with exceptions has not made it completely into the Isabelle 2001 release. Currently there is only the specification itself available. The proofs of soundness and completeness are broken (they still need to be ported to the exception version). Both theories are included for documentation (but they don't work for this specification), please see the Isabelle 99-2 release for a working copy.

types

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

consts

```
  merge :: "[jvm_prog, p_count, nat, nat, p_count, certificate, (nat × (state_type option))
list, state] ⇒ state"
```

primrec

```
  "merge G pc mxs mxr max_pc cert []     x = x"
  "merge G pc mxs mxr max_pc cert (s#ss) x = (let (pc',s') = s in
                                        if pc' < max_pc ∧ pc' = pc+1 then
                                          merge G pc mxs mxr max_pc cert ss (OK s' +_(sup G mxs
mxr) x)
                                        else if pc' < max_pc ∧ G ⊢ s' <=' cert!pc' then
                                          merge G pc mxs mxr max_pc cert ss x
                                        else Err)"
```

constdefs

```
  wtl_inst :: "[instr,jvm_prog,ty,state_type option,certificate,nat,nat,p_count,exception_table

              => state_type option err"
  "wtl_inst i G rT s cert maxs maxr max_pc et pc ==
     if app i G maxs rT pc et s then
       merge G pc maxs maxr max_pc cert (eff i G pc et s) (OK (cert!(pc+1)))
     else Err"

  wtl_cert :: "[instr,jvm_prog,ty,state_type option,certificate,nat,nat,p_count,exception_table

              => state_type option err"
  "wtl_cert i G rT s cert maxs maxr max_pc et pc ==
     case cert!pc of
        None    => wtl_inst i G rT s cert maxs maxr max_pc et pc
      | Some s' => if G ⊢ s <=' (Some s') then
                     wtl_inst i G rT (Some s') cert maxs maxr max_pc et pc
                   else Err"
```

consts

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

primrec

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

constdefs

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

```
 "wtl_method G C pTs rT mxs mxl et ins cert ==
       let max_pc = length ins
  in
  0 < max_pc ∧
  wtl_inst_list ins G rT cert mxs mxl max_pc et 0
                (Some ([],(OK (Class C))#((map OK pTs))@(replicate mxl Err))) ≠ Err"

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


lemmas [iff] = not_Err_eq

lemma if_eq_cases:
  "(P ⟹ x = z) ⟹ (¬P ⟹ y = z) ⟹ (if P then x else y) = z"
  by simp

lemma merge_def:
  "!!x. merge G pc mxs mxr max_pc cert ss x =
  (if ∀(pc',s') ∈ set ss. pc' < max_pc ∧ (pc' ≠ pc+1 ⟶ G ⊢ s' <=' cert!pc') then
    map (OK ∘ snd) [(p',t') ∈ ss. p'=pc+1] ++_(sup G mxs mxr) x
  else Err)" (is "!!x. ?merge ss x = ?if ss x" is "!!x. ?P ss x")
proof (induct ss)
  show "!!x. ?P [] x" by simp
next
  have OK_snd: "(λu. OK (snd u)) = OK ∘ snd" by (rule ext) auto
  fix x ss and s::"nat × (state_type option)"
  assume IH: "⋀x. ?P ss x"
  obtain pc' s' where s: "s = (pc',s')" by (cases s)
  hence "?merge (s#ss) x = ?merge ((pc',s')#ss) x" by hypsubst (rule refl)
  also
  have "... = (if pc' < max_pc ∧ pc' = pc+1 then
                ?merge ss (OK s' +_(sup G mxs mxr) x)
              else if pc' < max_pc ∧ G ⊢ s' <=' cert!pc' then
                ?merge ss x
              else Err)"
    (is "_ = (if ?pc' then ?merge ss (_ +_?f _) else if ?G then _ else _)")
    by simp
  also
  let "if ?all ss then _ else _" = "?if ss x"
  have "... = ?if ((pc',s')#ss) x"
  proof (cases "?pc'")
    case True
    hence "?all (((pc',s')#ss)) = (pc+1 < max_pc ∧ ?all ss)" by simp
    with True
    have "?if ss (OK s' +_?f x) = ?if ((pc',s')#ss) x" by (auto simp add: OK_snd)
    hence "?merge ss (OK s' +_?f x) = ?if ((pc',s')#ss) x" by (simp only: IH)
    with True show ?thesis by (fast intro: if_eq_cases)
  next
    case False
    have "(if ?G then ?merge ss x else Err) = ?if ((pc',s')#ss) x"
    proof (cases ?G)
      assume G: ?G with False
      have "?if ss x = ?if ((pc',s')#ss) x" by (auto simp add: OK_snd)
      hence "?merge ss x = ?if ((pc',s')#ss) x" by (simp only: IH)
      with G show ?thesis by (fast intro: if_eq_cases)
    next
```

```
    assume G: "¬?G"
    with False have "Err = ?if ((pc',s')#ss) x" by auto
    with G show ?thesis by (fast intro: if_eq_cases)
  qed
  with False show ?thesis by (fast intro: if_eq_cases)
 qed
 also from s have "... = ?if (s#ss) x" by hypsubst (rule refl)
 finally show "?P (s#ss) x" .
qed


lemma wtl_inst_OK:
"(wtl_inst i G rT s cert mxs mxr max_pc et pc = OK s') = (
 app i G mxs rT pc et s ∧
  (∀(pc',r) ∈ set (eff i G pc et s).
  pc' < max_pc ∧ (pc' ≠ pc+1 ⟶ G ⊢ r <=' cert!pc')) ∧
  map (OK ∘ snd) [(p',t') ∈ (eff i G pc et s). p'=pc+1]
  ++_(sup G mxs mxr) (OK (cert!(pc+1))) = OK s')"
 by (auto simp add: wtl_inst_def merge_def split: split_if_asm)

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

lemma wtl_append:
"∀ s pc. (wtl_inst_list (a@b) G rT cert mxs mxr mpc et pc s = OK s') =
   (∃s''. wtl_inst_list a G rT cert mxs mxr mpc et pc s = OK s'' ∧
          wtl_inst_list b G rT cert mxs mxr mpc et (pc+length a) s'' = OK s')"
 (is "∀ s pc. ?C s pc a" is "?P a")
proof (induct ?P a)
 show "?P []" by simp
next
 fix x xs  assume IH: "?P xs"
 show "?P (x#xs)"
 proof (intro allI)
  fix s pc
  show "?C s pc (x#xs)"
  proof (cases "wtl_cert x G rT s cert mxs mxr mpc et pc")
   case Err thus ?thesis by simp
  next
   fix s0 assume "wtl_cert x G rT s cert mxs mxr mpc et pc = OK s0"
   with IH have "?C s0 (Suc pc) xs" by blast
   thus ?thesis by (simp!)
  qed
 qed
qed

lemma wtl_take:
  "wtl_inst_list is G rT cert mxs mxr mpc et pc s = OK s'' ==>
   ∃s'. wtl_inst_list (take pc' is) G rT cert mxs mxr mpc et pc s = OK s'"
proof -
 assume "wtl_inst_list is G rT cert mxs mxr mpc et pc s = OK s''"
 hence "wtl_inst_list (take pc' is @ drop pc' is) G rT cert mxs mxr mpc et pc s = OK s''"
  by simp
 thus ?thesis by (auto simp add: wtl_append 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 wtl_Suc:
 "[| wtl_inst_list (take pc is) G rT cert maxs maxr (length is) et 0 s = OK s';
     wtl_cert (is!pc) G rT s' cert maxs maxr (length is) et pc = OK s'';
     Suc pc < length is |] ==>
  wtl_inst_list (take (Suc pc) is) G rT cert maxs maxr (length is) et 0 s = OK s''"
proof -
  assume "wtl_inst_list (take pc is) G rT cert maxs maxr (length is) et 0 s = OK s'"
         "wtl_cert (is!pc) G rT s' cert maxs maxr (length is) et pc = OK s''"
         "Suc pc < length is"
  hence "take (Suc pc) is = (take pc is)@[is!pc]" by (simp add: take_Suc)
  thus ?thesis by (simp! add: wtl_append min_def)
qed

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

  assume pc: "pc < length is"
  hence  "0 < length (drop pc is)" by simp
  then  obtain i r where Cons: "drop pc is = i#r"
    by (auto simp add: neq_Nil_conv simp del: length_drop)
  hence "i#r = drop pc is" ..
  with all have take:
    "wtl_inst_list (take pc is@i#r) G rT cert maxs maxr (length is) et 0 s ≠ Err"
    by simp

  from pc have "is!pc = drop pc is ! 0" by simp
  with Cons have "is!pc = i" by simp
  with take pc show ?thesis by (auto simp add: wtl_append min_def)
qed


end
```