theory *Basis = Main:*

ML_setup *{**
```
Unify.search_bound := 40;
Unify.trace_bound  := 40;

quick_and_dirty:=true;

Pretty.setmargin 198;
goals_limit:=2;
*}
```

1 misc

declare *same_fstI [intro!]*

ML *{**
```
fun make_simproc name pat pred thm = Simplifier.mk_simproc name
   [Thm.read_cterm (Thm.sign_of_thm thm) (pat, HOLogic.termT)]
   (K (K (fn s => if pred s then None else Some (standard (mk_meta_eq
thm)))))
*}
```

declare *split_if_asm [split] option.split [split] option.split_asm [split]*
ML *{**
```
simpset_ref() := simpset() addloop ("split_all_tac", split_all_tac)
*}
```
declare *if_weak_cong [cong del] option.weak_case_cong [cong del]*
declare *length_Suc_conv [iff]*

ML *{**
```
Delsimprocs [record_simproc];
*}
```

ML *{**
```
bind_thm ("make_imp", rearrange_prems [1,0] mp)
*}
```

lemma *Collect_split_eq: "{p. P (split f p)} = {(a,b). P (f a b)}"*
apply *auto*
done

lemma *subset_insertD:*
```
  "A <= insert x B ==> A <= B & x ~: A | (EX B'. A = insert x B' & B'
```

```
<= B)"
apply (case_tac "x:A")
apply (rule disjI2)
apply (rule_tac x = "A-{x}" in exI)
apply fast+
done

syntax
  "3" :: nat   ("3")
  "4" :: nat   ("4")
translations
 "3" == "Suc 2"
 "4" == "Suc 3"


lemma range_bool_domain: "range f = {f True, f False}"
apply auto
apply (case_tac "xa")
apply auto
done


lemma irrefl_tranclI': "r^-1 Int r^+ = {} ==> !x. (x, x) ~: r^+"
apply (rule allI)
apply (erule irrefl_tranclI)
done


lemma finite_SetCompr2: "[| finite (Collect P); !y. P y --> finite (range
(f y)) |] ==>
  finite {f y x |x y. P y}"
apply (subgoal_tac "{f y x |x y. P y} = UNION (Collect P) (%y. range (f
y))")
prefer 2 apply  fast
apply (erule ssubst)
apply (erule finite_UN_I)
apply fast
done


lemma list_all2_trans: "∀ a b c. P1 a b ⟶ P2 b c ⟶ P3 a c ⟹
 ∀xs2 xs3. list_all2 P1 xs1 xs2 ⟶ list_all2 P2 xs2 xs3 ⟶ list_all2
P3 xs1 xs3"
apply (induct_tac "xs1")
apply simp
apply (rule allI)
apply (induct_tac "xs2")
apply simp
apply (rule allI)
```

apply *(induct_tac "xs3")*
apply *auto*
done

2 pairs

lemma *surjective_pairing5: "p = (fst p, fst (snd p), fst (snd (snd p)),*
fst (snd (snd (snd p))),
snd (snd (snd (snd p))))"
apply *auto*
done

lemma *fst_splitE [elim!]:*
"[| fst s' = x'; !!x s. [| s' = (x,s); x = x' |] ==> Q |] ==> Q"
apply *(cut_tac p = "s'"* **in** *surjective_pairing)*
apply *auto*
done

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

3 quantifiers

ML *{**
fun noAll_simpset () = simpset() setmksimps
(mksimps (filter (fn (x,_) => x<>"All") mksimps_pairs)
**}*

lemma *All_Ex_refl_eq2 [simp]:*
"(!x. (? b. x = f b & Q b) ⟶ P x) = (!b. Q b --> P (f b))"
apply *auto*
done

lemma *ex_ex_miniscope1 [simp]:*
"(EX w v. P w v & Q v) = (EX v. (EX w. P w v) & Q v)"
apply *auto*
done

lemma *ex_miniscope2 [simp]:*
"(EX v. P v & Q & R v) = (Q & (EX v. P v & R v))"
apply *auto*
done

lemma *ex_reorder31: "(∃z x y. P x y z) = (∃x y z. P x y z)"*
apply *auto*

done

lemma *All_Ex_refl_eq1 [simp]: "(!x. (? b. x = f b) --> P x) = (!b. P (f b))"*
apply *auto*
done

4 sums

hide *const In0 In1*

syntax
fun_sum :: "('a => 'c) => ('b => 'c) => (('a+'b) => 'c)" (**infixr** *"'(+')"80)*
translations
"fun_sum" == "sum_case"

consts *the_Inl :: "'a + 'b ⇒ 'a"*
the_Inr :: "'a + 'b ⇒ 'b"
primrec *"the_Inl (Inl a) = a"*
primrec *"the_Inr (Inr b) = b"*

datatype *('a, 'b, 'c) sum3 = In1 'a | In2 'b | In3 'c*

consts *the_In1 :: "('a, 'b, 'c) sum3 ⇒ 'a"*
the_In2 :: "('a, 'b, 'c) sum3 ⇒ 'b"
the_In3 :: "('a, 'b, 'c) sum3 ⇒ 'c"
primrec *"the_In1 (In1 a) = a"*
primrec *"the_In2 (In2 b) = b"*
primrec *"the_In3 (In3 c) = c"*

syntax
In1l :: "'al ⇒ ('al + 'ar, 'b, 'c) sum3"
In1r :: "'ar ⇒ ('al + 'ar, 'b, 'c) sum3"
translations
"In1l e" == "In1 (Inl e)"
"In1r c" == "In1 (Inr c)"

ML *{**
```
fun sum3_instantiate thm = map (fn s => simplify(simpset()delsimps[not_None_eq])
 (read_instantiate [("t","In"^s^" ?x")] thm)) ["1l","2","3","1r"]
```
**}*

translations
"option"<= (type) "Option.option"
"list" <= (type) "List.list"
"sum3" <= (type) "Basis.sum3"

5 quantifiers for option type

syntax
```
  Oall :: "[pttrn, 'a option, bool] => bool"   ("(3! _:_:/ _)" [0,0,10]
10)
  Oex  :: "[pttrn, 'a option, bool] => bool"   ("(3? _:_:/ _)" [0,0,10]
10)
```

syntax *(symbols)*
```
  Oall :: "[pttrn, 'a option, bool] => bool"   ("(3∀ _∈_:/ _)"  [0,0,10]
10)
  Oex  :: "[pttrn, 'a option, bool] => bool"   ("(3∃ _∈_:/ _)"  [0,0,10]
10)
```

translations
```
  "! x:A: P"    == "! x:o2s A. P"
  "? x:A: P"    == "? x:o2s A. P"
```

6 unique association lists

constdefs
```
  unique   :: "('a × 'b) list ⇒ bool"
 "unique ≡ nodups ∘ map fst"
```

lemma *uniqueD [rule_format (no_asm)]:*
```
"unique l--> (!x y. (x,y):set l --> (!x' y'. (x',y'):set l --> x=x'-->
y=y'))"
```
apply *(unfold unique_def o_def)*
apply *(induct_tac "l")*
apply *(auto dest: fst_in_set_lemma)*
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

lemmas *unique_ConsI = conjI [THEN unique_Cons [THEN iffD2], standard]*

lemma *unique_single [simp]: "!!p. unique [p]"*
apply *auto*
done

lemma *unique_ConsD: "unique (x#xs) ==> unique xs"*
apply *(simp add: unique_def)*
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

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

7 list patterns

consts
lsplit :: "[['a, 'a list] => 'b, 'a list] => 'b"
defs
lsplit_def: "lsplit == %f l. f (hd l) (tl l)"

syntax
"_lpttrn" :: "[pttrn,pttrn] => pttrn" ("_#/_" [901,900] 900)
translations
"%y#x#xs. b" == "lsplit (%y x#xs. b)"
"%x#xs . b" == "lsplit (%x xs . b)"

lemma *lsplit [iff]: "lsplit c (x#xs) = c x xs"*
apply *(unfold lsplit_def)*
apply *(simp (no_asm))*
done

lemma *lsplit2: "lsplit P (x#xs) y z = P x xs y z"*
apply *(unfold lsplit_def)*
apply *simp*
done
declare *lsplit2 [iff]*

8 dummy pattern for quantifiers, let, etc.

syntax
```
  "@dummy_pat"   :: pttrn    ("'_")
```

parse_translation *{*
let fun dummy_pat_tr [] = Free ("_",dummyT)
  | dummy_pat_tr ts = raise TERM ("dummy_pat_tr", ts);
in [("@dummy_pat", dummy_pat_tr)]
end
}

end

theory *Name = Basis:*

typedecl *tnam*
typedecl *ename*
typedecl *mname*

arities
```
  tnam :: "term"
  ename :: "term"
  mname :: "term"
```

types *lname*
```
       = "ename + unit"
```
syntax
```
  EName :: lname
  This :: lname
```

translations
```
  "lname" <= (type) "ename + unit"
  "EName" => "Inl"
  "This"  => "Inr ()"
```

datatype *xname*
```
       = Throwable
       | NullPointer | OutOfMemory | ClassCast
       | NegArrSize  | IndOutBound | ArrStore
```

lemma *xn_cases:*
```
  "∀ xn. xn = Throwable   ∨ xn = NullPointer ∨
        xn = OutOfMemory ∨ xn = ClassCast ∨
        xn = NegArrSize  ∨ xn = IndOutBound ∨ xn = ArrStore"
```
apply *(rule allI)*

```
apply (induct_tac "xn")
apply auto
done

datatype tname
        = Object
        | SXcpt   xname
        | TName   tnam

translations
  "mname" <= (type) "Name.mname"
  "xname" <= (type) "Name.xname"
  "tname" <= (type) "Name.tname"
  "ename" <= (type) "Name.ename"

end



theory Type = Name:

datatype prim_ty
        = Void
        | Boolean
        | Integer


datatype ref_ty
        = NullT
        | IfaceT tname
        | ClassT tname
        | ArrayT ty

and ty
        = PrimT prim_ty
        | RefT  ref_ty

translations
  "prim_ty" <= (type) "Type.prim_ty"
  "ref_ty"  <= (type) "Type.ref_ty"
  "ty"      <= (type) "Type.ty"

syntax
         NT     :: "          ty"
         Iface  :: "tname  ⇒ ty"
         Class  :: "tname  ⇒ ty"
         Array  :: "ty     ⇒ ty"    ("_.[]" [90] 90)
```

```
translations
        "NT"      == "RefT   NullT"
        "Iface I" == "RefT (IfaceT I)"
        "Class C" == "RefT (ClassT C)"
        "T.[]"    == "RefT (ArrayT T)"

constdefs
  the_Class :: "ty ⇒ tname"
 "the_Class T ≡ εC. T = Class C"

lemma the_Class_eq [simp]: "the_Class (Class C)= C"
by (auto simp add: the_Class_def)

end



theory Value = Type:

typedecl loc
arities  loc :: "term"

datatype val
        = Unit
        | Bool bool
        | Intg int
        | Null
        | Addr loc


translations "val" <= (type) "Term.val"
             "loc" <= (type) "Term.loc"

consts   the_Bool   :: "val ⇒ bool"
primrec "the_Bool (Bool b) = b"
consts   the_Intg   :: "val ⇒ int"
primrec "the_Intg (Intg i) = i"
consts   the_Addr   :: "val ⇒ loc"
primrec "the_Addr (Addr a) = a"

types   dyn_ty      = "loc ⇒ ty option"
consts
  typeof        :: "dyn_ty ⇒ val ⇒ ty option"
  defpval       :: "prim_ty ⇒ val"
  default_val   :: "     ty ⇒ val"

primrec "typeof dt  Unit    = Some (PrimT Void)"
        "typeof dt (Bool b) = Some (PrimT Boolean)"
        "typeof dt (Intg i) = Some (PrimT Integer)"
```

```
        "typeof dt  Null    = Some NT"
        "typeof dt (Addr a) = dt a"

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



theory Term = Value:

datatype inv_mode
        = Static
        | SuperM
        | IntVir

types    sig
        = "mname × ty list"

datatype var
        = LVar                  lname
        | FVar  tname bool expr ename("{_,_}_.._"[10,10,85,99]90)
        | AVar        expr expr       ("_.[_]"[90,10   ]90)
and expr
        = NewC tname
        | NewA ty expr              ("New _[_]"[99,10   ]85)
        | Cast ty expr
        | Inst expr ref_ty              ("_ InstOf _"[85,99] 85)
        | Lit  val
        | Super
        | Acc  var
        | Ass  var expr             ("_:=_"   [90,85   ]85)
        | Cond expr expr expr        ("_ ? _ : _" [85,85,80]80)
        | Call ref_ty ref_ty inv_mode expr mname "(ty list)"
                "(expr list)" ("{_,_,_}_.._'( {_}_')"[10,10,10,85,99,10,10]85)
        | Methd tname sig
        | Body tname stmt expr
and  stmt
        = Skip
        | Expr  expr
        | Comp  stmt stmt          ("_;; _"                     [      66,65]65)
        | If_   expr stmt stmt     ("If'(_') _ Else _"          [   80,79,79]70)
        | Loop  expr stmt          ("While'(_') _"              [      80,79]70)
        | Throw expr
        | TryC  stmt
```

```
                tname ename stmt   ("Try _ Catch'(_ _') _"      [79,99,80,79]70)
        | Fin   stmt stmt          ("_ Finally _"              [      79,79]70)
        | init  tname
```

types *"term" = "(expr+stmt, var, expr list) sum3"*
translations

```
  "sig"   <= (type) "mname × ty list"
  "var"   <= (type) "Term.var"
  "expr"  <= (type) "Term.expr"
  "stmt"  <= (type) "Term.stmt"
  "term"  <= (type) "(expr+stmt, var, expr list) sum3"
```

ML *{**

```
fun sum3_instantiate thm = map (fn s => simplify(simpset()delsimps[not_None_eq])
 (read_instantiate [("t","In"^s^" ?x")] thm)) ["11","2","3","1r"];
```

**}*

syntax

```
  this    :: expr
  LAcc    :: "ename ⇒          expr" ("!!")
  LAss    :: "ename ⇒ expr ⇒ stmt" ("_:==_" [90,85] 85)
  StatRef :: "ref_ty ⇒ expr"
```

translations

```
 "this"       == "Acc (LVar This)"
 "!!v"        == "Acc (LVar (Inl v))"
 "v:==e"      == "Expr (Ass (LVar (Inl v)) e)"
 "StatRef rt" == "Cast (RefT rt) (Lit Null)"
```

constdefs

```
  is_stmt :: "term ⇒ bool"
 "is_stmt t ≡ ∃c. t=In1r c"
```

ML *{**

```
bind_thms ("is_stmt_rews", sum3_instantiate (thm "is_stmt_def"));
```

**}*
declare *is_stmt_rews [simp]*

end

theory *Table = Basis:*

types *('a, 'b) table*

```
      = "'a ⇝ 'b"
```

```
('a, 'b) tables
= "'a ⇒ 'b set"
```

9 $\mathbf{map}_o f / table_o f$

syntax
```
table_of      :: "('a × 'b) list ⇒ ('a, 'b) table"
```

translations
```
"table_of" == "map_of"

(type)"'a ⇝ 'b"       <= (type)"'a ⇒ 'b Option.option"
(type)"('a, 'b) table" <= (type)"'a ⇝ 'b"
```

lemma *Ball_set_table:* "(∀ (x,y)∈ set l. P x y) ⟹ ∀ x. ∀ y∈ map_of l x: P x y"
apply *(erule make_imp)*
apply *(induct l)*
apply *simp*
apply *(simp (no_asm))*
apply *auto*
done

lemma *Ball_set_tableD:*
```
"⟦(∀ (x,y)∈ set l. P x y); x ∈ o2s (table_of l xa)⟧ ⟹ P xa x"
```
apply *(frule Ball_set_table)*
by *auto*

declare *map_of_SomeD [elim]*

lemma *set_get_eq:*
```
"unique l ⟹ (k, the (table_of l k)) ∈ set l = (table_of l k ≠ None)"
```
apply *safe*
apply *(fast dest!: weak_map_of_SomeI)*
apply *auto*
done

lemma *inj_Pair_const2:* "inj (λk. (k, C))"
apply *(rule injI)*
apply *auto*
done

lemmas *table_of_map2_SomeI = inj_Pair_const2 [THEN map_of_mapk_SomeI, standard]*

lemma *table_of_map_SomeI [rule_format (no_asm)]:* "table_of t k = Some x ⟶
```
table_of (map (λ(k,x). (k, f x)) t) k = Some (f x)"
```

apply *(induct_tac "t")*
apply *auto*
done

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

lemma *table_of_mapf_Some [rule_format (no_asm)]: "∀x y. f x = f y ⟶*
x = y ⟹
table_of (map (λ(k,x). (k,f x)) t) k = Some (f x) ⟶ table_of t k
= Some x"
apply *(induct_tac "t")*
apply *auto*
done

lemma *table_of_mapf_SomeD [rule_format (no_asm), dest!]:*
"table_of (map (λ(k,x). (k, f x)) t) k = Some z ⟶ (∃y∈table_of t k:
z=f y)"
apply *(induct_tac "t")*
apply *auto*
done

lemma *table_of_mapkey_SomeD [rule_format (no_asm), dest!]:*
"table_of (map (λ(k,x). ((k,C),x)) t) (k,D) = Some x ⟶ C = D ∧ table_of
t k = Some x"
apply *(induct_tac "t")*
apply *auto*
done

lemma *table_append_Some_iff: "table_of (xs@ys) k = Some z =*
(table_of xs k = Some z ∨ (table_of xs k = None ∧ table_of ys k = Some
z))"
apply *(simp only: map_of_override [THEN sym])*
apply *(rule override_Some_iff)*
done

lemma *table_of_filter_unique_SomeD [rule_format (no_asm)]:*
"table_of (filter P xs) k = Some z ⟹ unique xs ⟶ table_of xs k
= Some z"
apply *(induct xs)*
apply *(auto del: map_of_SomeD intro!: map_of_SomeD)*
done

consts

```
  Un_tables      :: "('a, 'b) tables set ⇒ ('a, 'b) tables"
  overrides      :: "('a, 'b) tables     ⇒ ('a, 'b) tables ⇒
                     ('a, 'b) tables"             (infixl "⊕⊕" 100)
  hidings_entails:: "('a, 'b) tables ⇒ ('a, 'c) tables ⇒
                     ('b ⇒ 'c ⇒ bool) ⇒ bool"   ("_ hidings _ entails
_" 20)

  hiding_entails :: "('a, 'b) table  ⇒ ('a, 'c) table  ⇒
                     ('b ⇒ 'c ⇒ bool) ⇒ bool"   ("_ hiding _ entails
_"  20)
```

defs

```
  Un_tables_def:       "Un_tables ts    ≡ λk. ⋃t∈ts. t k"
  overrides_def:       "s ⊕⊕ t        ≡ λk. if t k = {} then s k else
t k"
  hidings_entails_def: "t hidings s entails R ≡ ∀k. ∀x∈t k. ∀y∈s k.
R x y"
  hiding_entails_def:  "t hiding  s entails R ≡ ∀k. ∀x∈t k: ∀y∈s k:
R x y"
```

10 Un$_t$*ables*

lemma *Un_tablesI [intro]:* "⋀x. ⟦t ∈ ts; x ∈ t k⟧ ⟹ x ∈ Un_tables
ts k"
apply *(simp add: Un_tables_def)*
apply *auto*
done

lemma *Un_tablesD [dest!]:* "⋀x. x ∈ Un_tables ts k ⟹ ∃t. t ∈ ts ∧
x ∈ t k"
apply *(simp add: Un_tables_def)*
apply *auto*
done

lemma *Un_tables_empty [simp]:* "Un_tables {} = (λk. {})"
apply *(unfold Un_tables_def)*
apply *(simp (no_asm))*
done

11 overrides

lemma *empty_overrides [simp]:* "(λk. {}) ⊕⊕ m = m"
apply *(unfold overrides_def)*
apply *(simp (no_asm))*
done
lemma *overrides_empty [simp]:* "m ⊕⊕ (λk. {}) = m"**apply** *(unfold overrides_def)*
apply *(simp (no_asm))*
done

lemma *overrides_Some_iff: "(x ∈ (s ⊕⊕ t) k) = (x ∈ t k ∨ t k = {} ∧ x ∈ s k)"*
by *(simp add: overrides_def)*

lemmas *overrides_SomeD = overrides_Some_iff [THEN iffD1, dest!]*

12 hiding$_e ntails$

lemma *hiding_entailsD: "⟦t hiding s entails R; t k = Some x; s k = Some y⟧ ⟹ R x y"*
by *(simp add: hiding_entails_def)*

lemma *empty_hiding_entails: "empty hiding s entails R"*
by *(simp add: hiding_entails_def)*

lemma *hiding_empty_entails: "t hiding empty entails R"*
by *(simp add: hiding_entails_def)*
declare *empty_hiding_entails [simp] hiding_empty_entails [simp]*

13 hidings$_e ntails$

lemma *hidings_entailsD: "⟦t hidings s entails R; x ∈ t k; y ∈ s k⟧ ⟹ R x y"*
by *(simp add: hidings_entails_def)*

lemma *hidings_empty_entails: "t hidings (λk. {}) entails R"*
apply *(unfold hidings_entails_def)*
apply *(simp (no_asm))*
done

lemma *empty_hidings_entails:*
*"(λk. {}) hidings s entails R"***apply** *(unfold hidings_entails_def)*
by *(simp (no_asm))*
declare *empty_hidings_entails [intro!] hidings_empty_entails [intro!]*

consts
atleast_free :: "('a ~=> 'b) => nat => bool"
primrec
"atleast_free m 0 = True"
atleast_free_Suc:
"atleast_free m (Suc n) = (? a. m a = None & (!b. atleast_free (m(a|->b)) n))"

lemma *atleast_free_weaken [rule_format (no_asm)]:*

```
  "!m. atleast_free m (Suc n) ⟶ atleast_free m n"
apply (induct_tac "n")
apply (simp (no_asm))
apply clarify
apply (simp (no_asm))
apply (drule atleast_free_Suc [THEN iffD1])
apply fast
done

lemma atleast_free_SucI:
"[| h a = None; !obj. atleast_free (h(a|->obj)) n |] ==> atleast_free
h (Suc n)"
by force

declare fun_upd_apply [simp del]
lemma atleast_free_SucD_lemma [rule_format (no_asm)]:
" !m a. m a = None --> (!c. atleast_free (m(a|->c)) n) -->
  (!b d. a ~= b --> atleast_free (m(b|->d)) n)"
apply (induct_tac "n")
apply  auto
apply (rule_tac x = "a" in exI)
apply  (rule conjI)
apply  (force simp add: fun_upd_apply)
apply (erule_tac V = "m a = None" in thin_rl)
apply clarify
apply (subst fun_upd_twist)
apply  (erule not_sym)
apply (rename_tac "ba")
apply (drule_tac x = "ba" in spec)
apply clarify
apply (tactic "smp_tac 2 1")
apply (erule (1) notE impE)
apply (case_tac "aa = b")
apply fast+
done
declare fun_upd_apply [simp]

lemma atleast_free_SucD [rule_format (no_asm)]: "atleast_free h (Suc
n) ==> atleast_free (h(a|->b)) n"
apply auto
apply (case_tac "aa = a")
apply auto
apply (erule atleast_free_SucD_lemma)
apply auto
done

declare atleast_free_Suc [simp del]

end
```

theory *Decl = Term + Table:*

types *modi = bool*

field
= "modi × ty"

fdecl
= "ename × field"

translations
"field" <= (type) "bool × ty"
"fdecl" <= (type) "ename × field"

types

mhead
= "modi × ename list × ty"

mbody
= "(ename × ty) list × stmt × expr"

methd
= "mhead × mbody"

mdecl
= "sig × methd"

translations
"mhead" <= (type) "bool × ename list × ty"
"mbody" <= (type) "(ename × ty) list × stmt × expr"
"methd" <= (type) "mhead × mbody"
"mdecl" <= (type) "sig × methd"

syntax

static :: "modi ⇒ bool"
mrt :: "mhead ⇒ ty"

translations
"static" => "id"
"mrt mh" => "snd (snd mh)"

types *ibody*
= "(sig × mhead) list"

iface
= "tname list × ibody"

idecl
= "tname × iface"

translations
"ibody" <= (type) "(sig × mhead) list"
"iface" <= (type) "tname list × ibody"
"idecl" <= (type) "tname × iface"

types *cbody*
= "fdecl list × mdecl list × stmt"

class
= "tname × tname list × cbody"

cdecl
= "tname × class"

translations
"cbody" <= (type) "fdecl list × mdecl list × stmt"
"class" <= (type) "tname × tname list × cbody"
"cdecl" <= (type) "tname × class"

14 standard classes

consts

Object_mdecls :: "mdecl list"
SXcpt_mdecls :: "mdecl list"
ObjectC :: "cdecl"
SXcptC ::"xname ⇒ cdecl"

defs

ObjectC_def: "ObjectC ≡ (Object , (arbitrary ,[],[],Object_mdecls,Skip))"
SXcptC_def: "SXcptC xn≡ (SXcpt xn, (if xn = Throwable then Object else
SXcpt Throwable,[],[],SXcpt_mdecls,Skip))"
lemma *ObjectC_neq_SXcptC [simp]: "ObjectC ≠ SXcptC xn"*

by *(simp add: ObjectC_def SXcptC_def)*

lemma *SXcptC_inject [simp]: "(SXcptC xn = SXcptC xm) = (xn = xm)"*
apply *(simp add: SXcptC_def)*
apply *auto*
done

constdefs *standard_classes :: "cdecl list"*
"standard_classes ≡ [ObjectC, SXcptC Throwable,
SXcptC NullPointer, SXcptC OutOfMemory, SXcptC ClassCast,
SXcptC NegArrSize , SXcptC IndOutBound, SXcptC ArrStore]"

15 programs

types *prog = "idecl list × cdecl list"*
translations
"prog"<= (type) "idecl list × cdecl list"

syntax
iface :: "prog ⇒ (tname, iface) table"
class :: "prog ⇒ (tname, class) table"
is_iface :: "prog ⇒ tname ⇒ bool"
is_class :: "prog ⇒ tname ⇒ bool"

translations
"iface G I" == "table_of (fst G) I"
"class G C" == "table_of (snd G) C"
"is_iface G I" == "iface G I ≠ None"
"is_class G C" == "class G C ≠ None"

16 is$_{type}$

consts
is_type :: "prog ⇒ ty ⇒ bool"
isrtype :: "prog ⇒ ref_ty ⇒ bool"

primrec *"is_type G (PrimT pt) = True"*
"is_type G (RefT rt) = isrtype G rt"
"isrtype G (NullT) = True"
"isrtype G (IfaceT tn) = is_iface G tn"
"isrtype G (ClassT tn) = is_class G tn"
"isrtype G (ArrayT T) = is_type G T"

lemma *type_is_iface: "is_type G (Iface I) ⟹ is_iface G I"*
by *auto*

lemma *type_is_class: "is_type G (Class C) ⟹ is_class G C"*
by *auto*

17 subinterface and subclass relation, in anticipation of TypeRel.thy

consts

```
  subint1  :: "prog ⇒ (tname × tname) set"
  subcls1  :: "prog ⇒ (tname × tname) set"
```

defs

```
  subint1_def:   "subint1 G ≡ {(I,J). ∃i∈iface G I: J∈set (fst i)}"
  subcls1_def:   "subcls1 G ≡ {(C,D). C≠Object ∧ (∃c∈class G C: fst
c = D)}"
```

lemma *subint1I:* `"⟦iface G I = Some (si,ms); J ∈ set si⟧ ⟹ (I,J) ∈ subint1 G"`
apply *(simp add: subint1_def)*
done

lemma *subcls1I:* `"⟦class G C = Some (D,rest); C ≠ Object⟧ ⟹ (C,D) ∈ subcls1 G"`
apply *(simp add: subcls1_def)*
done

lemma *subint1D:* `"(I,J)∈subint1 G⟹ ∃(si,ms)∈iface G I: J∈set si"`
apply *(simp add: subint1_def)*
apply *auto*
done

lemma *subcls1D:* `"(C,D)∈subcls1 G ⟹ C≠Object ∧ (∃cb. class G C = Some (D,cb))"`
apply *(simp add: subcls1_def)*
apply *auto*
done

lemma *subint1_def2:*

```
   "subint1 G = (Σ I∈{I. is_iface G I}. set (fst (the (iface G I))))"
```

apply *(unfold subint1_def)*
apply *auto*
done

lemma *subcls1_def2:*

```
   "subcls1 G = (ΣC∈{C. is_class G C}. {D. C≠Object ∧ fst(the(class
G C))=D})"
```

apply *(unfold subcls1_def)*
apply *auto*
done

18 well-structured programs

constdefs

```
  ws_idecl :: "prog ⇒ tname ⇒ tname list ⇒ bool"
 "ws_idecl G I si ≡ ∀J∈set si.  is_iface G J   ∧ (J,I)∉(subint1 G)^+"

  ws_cdecl :: "prog ⇒ tname ⇒ tname ⇒ bool"
 "ws_cdecl G C sc ≡ C≠Object ⟶ is_class G sc ∧ (sc,C)∉(subcls1 G)^+"

  ws_prog  :: "prog ⇒ bool"
 "ws_prog G ≡ (∀(I,(si,ib))∈set (fst G). ws_idecl G I si) ∧
              (∀(C,(sc,cb))∈set (snd G). ws_cdecl G C sc)"
```

lemma *ws_progI:*

```
"⟦∀(I,si,ib)∈set (fst G). ∀J∈set si.   is_iface G J ∧ (J,I) ∉ (subint1
G)^+;
  ∀(C,D ,cb)∈set (snd G). C≠Object ⟶ is_class G D ∧ (D,C) ∉ (subcls1
G)^+
 ⟧ ⟹ ws_prog G"
```

apply *(unfold ws_prog_def ws_idecl_def ws_cdecl_def)*
apply *(erule_tac conjI)*
apply *blast*
done

lemma *ws_prog_ideclD:*

```
"⟦iface G I = Some (si,ib); J∈set si; ws_prog G⟧ ⟹
  is_iface G J ∧ (J,I)∉(subint1 G)^+"
```

apply *(unfold ws_prog_def ws_idecl_def)*
apply *clarify*
apply *(drule_tac map_of_SomeD)*
apply *auto*
done

lemma *ws_prog_cdeclD:*

```
"⟦class G C = Some(sc,cb); C≠Object; ws_prog G⟧ ⟹
  is_class G sc ∧ (sc,C)∉(subcls1 G)^+"
```

apply *(unfold ws_prog_def ws_cdecl_def)*
apply *clarify*
apply *(drule_tac map_of_SomeD)*
apply *auto*
done

19 well-foundedness

lemma *finite_is_iface: "finite {I. is_iface G I}"*
apply *(fold dom_def)*
apply *(rule_tac finite_dom_map_of)*
done

```
lemma finite_is_class: "finite {C. is_class G C}"
apply (fold dom_def)
apply (rule_tac finite_dom_map_of)
done

lemma finite_subint1: "finite (subint1 G)"
apply (subst subint1_def2)
apply (rule finite_SigmaI)
apply (rule finite_is_iface)
apply (simp (no_asm))
done

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

lemma subint1_irrefl_lemma1: "ws_prog G ⟹ (subint1 G)^-1 ∩ (subint1
G)^+ = {}"
apply (force dest: subint1D ws_prog_ideclD conjunct2)
done

lemma subcls1_irrefl_lemma1: "ws_prog G ⟹ (subcls1 G)^-1 ∩ (subcls1
G)^+ = {}"
apply (force dest: subcls1D ws_prog_cdeclD conjunct2)
done

lemmas subint1_irrefl_lemma2 = subint1_irrefl_lemma1 [THEN irrefl_tranclI']
lemmas subcls1_irrefl_lemma2 = subcls1_irrefl_lemma1 [THEN irrefl_tranclI']

lemma subint1_irrefl: "⟦(x, y) ∈ subint1 G; ws_prog G⟧ ⟹ x ≠ y"
apply (rule irrefl_trancl_rD)
apply (rule subint1_irrefl_lemma2)
apply auto
done

lemma subcls1_irrefl: "⟦(x, y) ∈ subcls1 G; ws_prog G⟧ ⟹ x ≠ y"
apply (rule irrefl_trancl_rD)
apply (rule subcls1_irrefl_lemma2)
apply auto
done

lemmas subint1_acyclic = subint1_irrefl_lemma2 [THEN acyclicI, standard]
lemmas subcls1_acyclic = subcls1_irrefl_lemma2 [THEN acyclicI, standard]
```

lemma *wf_subint1: "ws_prog G ⟹ wf ((subint1 G)⁻¹)"*
by *(auto intro: finite_acyclic_wf_converse finite_subint1 subint1_acyclic)*

lemma *wf_subcls1: "ws_prog G ⟹ wf ((subcls1 G)⁻¹)"*
by *(auto intro: finite_acyclic_wf_converse finite_subcls1 subcls1_acyclic)*

lemma *subint1_induct:*
"⟦ws_prog G; ⋀x. ∀y. (x, y) ∈ subint1 G ⟶ P y ⟹ P x⟧ ⟹ P a"
apply *(frule wf_subint1)*
apply *(erule wf_induct)*
apply *(simp (no_asm_use) only: converse_iff)*
apply *blast*
done

lemma *subcls1_induct:*
"⟦ws_prog G; ⋀x. ∀y. (x, y) ∈ subcls1 G ⟶ P y ⟹ P x⟧ ⟹ P a"
apply *(frule wf_subcls1)*
apply *(erule wf_induct)*
apply *(simp (no_asm_use) only: converse_iff)*
apply *blast*
done

lemma *ws_subint1_induct: "⟦is_iface G I; ws_prog G;*
⋀I is ms. ⟦iface G I = Some (is, ms) ∧
(∀J ∈ set is. (I,J)∈subint1 G ∧ P J ∧ is_iface G J)⟧ ⟹ P I⟧ ⟹
P I"
apply *(cut_tac)*
apply *(erule make_imp)*
apply *(rule subint1_induct)*
apply *assumption*
apply *safe*
apply *(fast dest: subint1I ws_prog_ideclD)*
done

lemma *ws_subcls1_induct: "⟦is_class G C; ws_prog G;*
⋀C D si fs ms ini. ⟦class G C = Some (D,si,fs,ms,ini) ∧
(C ≠ Object ⟶ (C,D)∈subcls1 G ∧ P D ∧ is_class G D)⟧ ⟹ P C
⟧ ⟹ P C"
apply *(cut_tac)*
apply *(erule make_imp)*
apply *(rule subcls1_induct)*
apply *assumption*
apply *safe*
apply *(fast dest: subcls1I ws_prog_cdeclD)*
done

20 general recursion operators for the interface and class hiearchies

consts
```
  iface_rec   :: "prog × tname ⇒        (tname ⇒ ibody ⇒ 'a set ⇒
'a) ⇒ 'a"
  class_rec   :: "prog × tname ⇒ 'a ⇒ (tname ⇒ cbody ⇒ 'a     ⇒
'a) ⇒ 'a"

recdef iface_rec "same_fst ws_prog (λG. (subint1 G)^-1)"
"iface_rec (G,I) = (λf. case iface G I of None ⇒ arbitrary | Some (si,ib)
⇒
                    if ws_prog G then f I ib ((λJ. iface_rec (G,J) f)`set
si)
                    else arbitrary)"
(hints recdef_wf: wf_subint1 intro: subint1I)
declare iface_rec.simps [simp del]

lemma iface_rec: " ⟦iface G I = Some (si,ib); ws_prog G⟧ ⟹
  iface_rec (G,I) f = f I ib ((λJ. iface_rec (G,J) f)`set si)"
apply (subst iface_rec.simps)
apply simp
done

recdef class_rec "same_fst ws_prog (λG. (subcls1 G)^-1)"
"class_rec(G,C) = (λt f. case class G C of None⇒ arbitrary | Some (sc,si,cb)⇒
       if ws_prog G then f C cb (if C = Object then t else class_rec (G,sc)
t f)
       else arbitrary)"
(hints recdef_wf: wf_subcls1 intro: subcls1I)
declare class_rec.simps [simp del]

lemma class_rec: "⟦class G C = Some (sc,si,cb); ws_prog G⟧ ⟹
 class_rec (G,C) t f = f C cb (if C = Object then t else class_rec (G,sc)
t f)"
apply (rule class_rec.simps [THEN trans [THEN fun_cong [THEN fun_cong]]])
apply simp
done
```

21 fields and methods

types
```
  fspec = "ename × tname"
```
consts
```
  imethds       :: "prog ⇒ tname ⇒ ( sig   , tname  ×  mhead) tables"
  cmethd        :: "prog ⇒ tname ⇒ ( sig   , tname  ×  methd) table"
  fields        :: "prog ⇒ tname ⇒ ((ename × tname) ×  field) list"
  cfield        :: "prog ⇒ tname ⇒ ( ename , tname  ×  field) table"
```

defs

```
  imeths_def: "imethds G I ≡ iface_rec (G,I)       (λI     ms      ts.

       (Un_tables ts) ⊕⊕ (o2s ∘ table_of (map (λ(s,m). (s,I,m)) ms)))"


  cmethd_def: "cmethd  G C ≡ class_rec (G,C) empty (λC (fs,ms,ini) ts.
                             ts ++ table_of (map (λ(s,m). (s,C,m)) ms))"


  fields_def: "fields G C ≡ class_rec (G,C) []    (λC (fs,ms,ini) ts.
                                          map (λ(n,t). ((n,C),t)) fs
@ ts)"


  cfield_def: "cfield G C ≡ table_of((map (λ((n,d),T).(n,(d,T)))) (fields
G C))"
```

constdefs

```
  is_methd :: "prog ⇒ tname ⇒ sig ⇒ bool"
 "is_methd G ≡ λC sig. is_class G C ∧ cmethd G C sig ≠ None"
```

21.1 imethds

lemma *imethds_rec:* "⟦*iface G I = Some (is,ms); ws_prog G*⟧ ⟹
  imethds G I = Un_tables ((λJ. imethds G J)‘set is) ⊕⊕
                          (o2s ∘ table_of (map (λ(s,mh). (s,I,mh)) ms))"
apply *(unfold imeths_def)*
apply *(rule iface_rec [THEN trans])*
apply *auto*
done

lemma *imethds_norec:*
  "⟦*iface G md = Some (is, ms); ws_prog G; table_of ms sig = Some mh*⟧
⟹
  (md, mh) ∈ imethds G md sig"
apply *(subst imethds_rec)*
apply *assumption+*
apply *(rule iffD2)*
apply *(rule overrides_Some_iff)*
apply *(rule disjI1)*
apply *(auto elim: table_of_map_SomeI)*
done

lemma *imethds_defpl: "⟦(md,mh) ∈ imethds G I sig; ws_prog G; is_iface G I⟧ ⟹*
(∃is ms. iface G md = Some (is, ms) ∧ table_of ms sig = Some mh) ∧

(I,md) ∈ (subint1 G)^* ∧ (md,mh) ∈ imethds G md sig"
apply *(erule make_imp)*
apply *(rule ws_subint1_induct, assumption, assumption)*
apply *(subst imethds_rec, erule conjunct1, assumption)*
apply *(force elim: imethds_norec intro: rtrancl_into_rtrancl2)*
done

21.2 cmethd

lemma *cmethd_rec: "⟦class G C = Some (sc,si,fs,ms,ini); ws_prog G⟧ ⟹*

cmethd G C = (if C = Object then empty else cmethd G sc) ++
table_of (map (λ(s,m). (s,(C,m))) ms)"
apply *(unfold cmethd_def)*
apply *(erule class_rec [THEN trans], assumption)*
apply *clarsimp*
done

lemma *cmethd_norec: "⟦class G md = Some (D, si, fs, ms, ini); ws_prog G;*
table_of ms sig = Some m⟧ ⟹ cmethd G md sig = Some (md, m)"
apply *(subst cmethd_rec)*
apply *assumption+*
apply *(rule disjI1 [THEN override_Some_iff [THEN iffD2]])*
apply *(auto elim: table_of_map_SomeI)*
done

lemma *cmethd_defpl: "⟦cmethd G C sig = Some (md, m); ws_prog G; is_class G C⟧ ⟹*
(∃D si fs ms ini. class G md=Some (D,si,fs,ms,ini) ∧ table_of ms sig=Some m) ∧
(C,md)∈(subcls1 G)^* ∧ cmethd G md sig = Some (md, m)"
apply *(erule make_imp)*
apply *(rule ws_subcls1_induct, assumption, assumption)*
apply *(subst cmethd_rec, erule conjunct1, assumption)*
apply *(force elim: cmethd_norec intro: rtrancl_into_rtrancl2)*
done

lemma *finite_cmethd:"ws_prog G ⟹ finite {cmethd G C sig |sig C. is_class G C}"*
apply *(rule finite_is_class [THEN finite_SetCompr2])*
apply *(intro strip)*

```
apply (erule_tac ws_subcls1_induct, assumption)
apply (subst cmethd_rec)
apply (erule_tac conjunct1, assumption)
apply (auto intro!: finite_range_map_of elim!: finite_range_map_of_override)
done

lemma finite_dom_cmethd:"⟦ws_prog G; is_class G C⟧ ⟹ finite (dom (cmethd
G C))"
apply (erule_tac ws_subcls1_induct)
apply assumption
apply (subst cmethd_rec)
apply (erule_tac conjunct1, assumption)
apply (auto intro!: finite_dom_map_of)
done
```

21.3 fields

```
lemma fields_rec: "⟦class G C = Some (sc,si,fs,ms,ini); ws_prog G⟧ ⟹

  fields G C = map (λ(fn,ft). ((fn,C),ft)) fs @
  (if C = Object then [] else fields G sc)"
apply (simp only: fields_def)
apply (erule class_rec [THEN trans])
apply assumption
apply clarsimp
done


lemma fields_norec: "⟦class G fd = Some (D, si, fs, ms, ini); ws_prog
G;
  table_of fs fn = Some f⟧ ⟹ table_of (fields G fd) (fn,fd) = Some f"
apply (subst fields_rec)
apply assumption+
apply (subst map_of_override [symmetric])
apply (rule disjI1 [THEN override_Some_iff [THEN iffD2]])
apply (auto elim: table_of_map2_SomeI)
done

lemma fields_defpl:
 "⟦table_of (fields G C) (fn,fd) = Some f; ws_prog G; is_class G C⟧ ⟹

  (∃D si fs ms ini. class G fd=Some (D,si,fs,ms,ini) ∧ table_of fs fn=Some
f) ∧
  (C,fd)∈(subcls1 G)^* ∧ table_of (fields G fd) (fn,fd) = Some f"
apply (erule make_imp)
apply (rule ws_subcls1_induct, assumption, assumption)
apply (subst fields_rec, erule conjunct1, assumption)
apply (auto elim: fields_norec intro: rtrancl_into_rtrancl2
            simp add: map_of_override [symmetric] simp del: map_of_override)
```

done

lemma *fields_emptyI: "⋀y. ⟦ws_prog G; class G C = Some (sc, is, [], m, c); C ≠ Object ⟶ class G sc = Some y ∧ fields G sc = []⟧ ⟹ fields G C = []"*
apply *(subst fields_rec)*
apply *assumption*
apply *auto*
done

lemma *fields_mono_lemma: "⟦x ∈ set (fields G C); (D,C)∈(subcls1 G)^*; ws_prog G⟧ ⟹ x ∈ set (fields G D)"*
apply *(erule make_imp)*
apply *(erule converse_rtrancl_induct)*
apply *fast*
apply *(drule subcls1D)*
apply *clarsimp*
apply *(subst fields_rec)*
apply *auto*
done

lemma *ws_unique_fields_lemma: "⟦((fn, C), f1) ∈ set (fields G D); (fn, f2) ∈ set fs; ws_prog G; class G C = Some (D, si, fs, m_i); C ≠ Object; class G D = Some c⟧ ⟹ R"*
apply *(frule_tac ws_prog_cdeclD [THEN conjunct2], assumption, assumption)*
apply *(drule_tac weak_map_of_SomeI)*
apply *(frule_tac subcls1I [THEN subcls1_irrefl], assumption, assumption)*
apply *(auto dest: fields_defpl [THEN conjunct2 [THEN conjunct1[THEN rtranclD]]])*
done

lemma *ws_unique_fields: "⟦is_class G C; ws_prog G; ⋀C D s fs r. ⟦class G C = Some (D, s, fs, r)⟧ ⟹ unique fs ⟧ ⟹ unique (fields G C)"*
apply *cut_tac*
apply *(rule ws_subcls1_induct, assumption, assumption)*
apply *(subst fields_rec, erule conjunct1, assumption)*
apply *(auto intro!: unique_map_inj injI elim!: unique_append ws_unique_fields_lemma fields_norec)*
done

21.4 cfield

lemma *cfield_fields:*
"cfield G C fn = Some (fd, fT) ⟹ table_of (fields G C) (fn, fd) = Some fT"
apply *(simp only: cfield_def)*
apply *(rule table_of_remap_SomeD)*
apply *simp*
done

lemma *cfield_defpl_is_class:*
"⟦is_class G C; cfield G C en = Some (fd, b, fT); ws_prog G⟧ ⟹ is_class G fd"
apply *(drule cfield_fields)*
apply *(drule fields_defpl [THEN conjunct1], assumption)*
apply *auto*
done

21.5 is$_m$*ethd*

lemma *is_methdI:*
"⟦class G C = Some y; cmethd G C sig = Some b⟧ ⟹ is_methd G C sig"
apply *(unfold is_methd_def)*
apply *auto*
done

lemma *is_methdD:*
"is_methd G C sig ⟹ class G C ≠ None ∧ cmethd G C sig ≠ None"
apply *(unfold is_methd_def)*
apply *auto*
done

lemma *finite_is_methd: "ws_prog G ⟹ finite (Collect (split (is_methd G)))"*
apply *(unfold is_methd_def)*
apply *(subst SetCompr_Sigma_eq)*
apply *(rule finite_is_class [THEN finite_SigmaI])*
apply *(simp only: mem_Collect_eq)*
apply *(fold dom_def)*
apply *(erule finite_dom_cmethd)*
apply *assumption*
done

end

theory *TypeRel = Decl:*

consts

```
  implmt1   :: "prog ⇒ (tname × tname) set"
  implmt    :: "prog ⇒ (tname × tname) set"
  widen     :: "prog ⇒ (ty    × ty   ) set"
  narrow    :: "prog ⇒ (ty    × ty   ) set"
  cast     :: "prog ⇒ (ty    × ty   ) set"
```

syntax

```
 "@subint1" :: "prog => [tname, tname] => bool" ("_|-_<:I1_" [71,71,71]
70)
 "@subint"  :: "prog => [tname, tname] => bool" ("_|-_<=:I _"[71,71,71]
70)
 "@subcls1" :: "prog => [tname, tname] => bool" ("_|-_<:C1_" [71,71,71]
70)
 "@subcls"  :: "prog => [tname, tname] => bool" ("_|-_<=:C _"[71,71,71]
70)
 "@implmt1" :: "prog => [tname, tname] => bool" ("_|-_~>1_"  [71,71,71]
70)
 "@implmt"  :: "prog => [tname, tname] => bool" ("_|-_~>_"   [71,71,71]
70)
 "@widen"   :: "prog => [ty   , ty   ] => bool" ("_|-_<=:_"  [71,71,71]
70)
 "@narrow"  :: "prog => [ty   , ty   ] => bool" ("_|-_:>_"   [71,71,71]
70)
 "@cast"    :: "prog => [ty   , ty   ] => bool" ("_|-_<=:? _"[71,71,71]
70)
```

syntax *(symbols)*

```
  "@subint1" :: "prog ⇒ [tname, tname] ⇒ bool" ("_⊢_≺I1_"  [71,71,71]
70)
  "@subint"  :: "prog ⇒ [tname, tname] ⇒ bool" ("_⊢_≼I _"  [71,71,71]
70)
  "@subcls1" :: "prog ⇒ [tname, tname] ⇒ bool" ("_⊢_≺C1_"  [71,71,71]
70)
  "@subcls"  :: "prog ⇒ [tname, tname] ⇒ bool" ("_⊢_≼C _"  [71,71,71]
70)
  "@implmt1" :: "prog ⇒ [tname, tname] ⇒ bool" ("_⊢_⇝1_"  [71,71,71]
70)
  "@implmt"  :: "prog ⇒ [tname, tname] ⇒ bool" ("_⊢_⇝_"   [71,71,71]
70)
  "@widen"   :: "prog ⇒ [ty   , ty   ] ⇒ bool" ("_⊢_≼_"    [71,71,71]
70)
  "@narrow"  :: "prog ⇒ [ty   , ty   ] ⇒ bool" ("_⊢_≻_"    [71,71,71]
```

```
70)
  "@cast"    :: "prog ⇒ [ty   , ty   ] ⇒ bool" ("_⊢_≼? _"  [71,71,71]
70)
```

translations

```
"G⊢I ≺I1 J" == "(I,J) ∈ subint1 G"
"G⊢I ≼I  J" == "(I,J) ∈(subint1 G)^*"
"G⊢C ≺C1 D" == "(C,D) ∈ subcls1 G"
"G⊢C ≼C  D" == "(C,D) ∈(subcls1 G)^*"
"G⊢C ↝1 I" == "(C,I) ∈ implmt1 G"
"G⊢C ↝  I" == "(C,I) ∈ implmt  G"
"G⊢S ≼   T" == "(S,T) ∈ widen   G"
"G⊢S ≻   T" == "(S,T) ∈ narrow  G"
"G⊢S ≼?  T" == "(S,T) ∈ cast    G"
```

22 subclass and subinterface relations

lemmas *subcls_direct = subcls1I [THEN r_into_rtrancl, standard]*

lemma *SXcpt_subcls_Throwable_lemma: "⟦class G (SXcpt xn) = Some (if xn = Throwable then Object else SXcpt Throwable, rest)⟧ ⟹ G⊢SXcpt xn≼C SXcpt Throwable"*
apply *(case_tac "xn = Throwable")*
apply *simp_all*
apply *(erule subcls_direct)*
apply *(simp (no_asm))*
done

lemma *subcls_ObjectI: "⟦is_class G C; ws_prog G⟧ ⟹ G⊢C≼C Object"*
apply *(erule ws_subcls1_induct)*
apply *clarsimp*
apply *(case_tac "C = Object")*
apply *(fast intro: r_into_rtrancl [THEN rtrancl_trans])+*
done

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

lemma *subcls_is_class: "(C,D) ∈ (subcls1 G)^+ ⟹ is_class G C"*
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 *(erule rtrancl_induct)*
apply *(drule_tac [2] subcls1D)*
apply *auto*
done

23 implementation relation

defs

implmt1_def:"implmt1 G≡{(C,I). C≠Object ∧ (∃c∈class G C: I∈set (fst(snd c)))}"

lemma *implmt1D: "G⊢C⇝1I ⟹ C≠Object ∧ (∃(sc,is,rest)∈class G C: I∈set is)"*
apply *(unfold implmt1_def)*
apply *auto*
done

inductive *"implmt G"* **intros**

direct: *"G⊢C⇝1J ⟹ G⊢C⇝J"*
subint: *"⟦G⊢C⇝1I; G⊢I≼I J⟧ ⟹ G⊢C⇝J"*
subcls1: *"⟦G⊢C≺C1D; G⊢D⇝J ⟧ ⟹ G⊢C⇝J"*

lemma *implmtD: "G⊢C⇝J ⟹ (∃I. G⊢C⇝1I ∧ G⊢I≼I J) ∨ (∃D. G⊢C≺C1D ∧ G⊢D⇝J)"*
apply *(erule implmt.induct)*
apply *fast+*
done

lemma *implmt_ObjectE [elim!]: "G⊢Object⇝I ⟹ R"*
by *(auto dest!: implmtD implmt1D subcls1D)*

lemma *subcls_implmt [rule_format (no_asm)]: "G⊢A≼C B ⟹ G⊢B⇝K ⟶ G⊢A⇝K"*
apply *(erule rtrancl_induct)*
apply *(auto intro: implmt.subcls1)*
done

lemma *implmt_subint2: "⟦ G⊢A⇝J; G⊢J≼I K⟧ ⟹ G⊢A⇝K"*
apply *(erule make_imp, erule implmt.induct)*
apply *(auto dest: implmt.subint rtrancl_trans implmt.subcls1)*
done

lemma *implmt_is_class: "G⊢C⇝I ⟹ is_class G C"*
apply *(erule implmt.induct)*
apply *(blast dest: implmt1D subcls1D)+*
done

24 widening relation

inductive *"widen G"* **intros**

```
  refl:                          "G⊢        T≼T"
  subint:  "G⊢I≼I J"          ⟹ G⊢  Iface I≼ Iface J"
  int_obj:                         "G⊢  Iface I≼ Class Object"
  subcls:  "G⊢C≼C D"           ⟹ G⊢  Class C≼ Class D"
  implmt:  "G⊢C⇝I"             ⟹ G⊢  Class C≼ Iface I"
  null:                            "G⊢       NT≼ RefT R"
  arr_obj:                         "G⊢     T.[]≼ Class Object"
  array:   "G⊢RefT S≼RefT T ⟹ G⊢RefT S.[]≼ RefT T.[]"
```

declare *widen.refl [intro!]*
declare *widen.intros [simp]*

lemma *widen_PrimT: "G⊢PrimT x≼T ⟹ (∃y. T = PrimT y)"*
apply *(ind_cases "G⊢S≼T")*
by *auto*

lemma *widen_PrimT2: "G⊢S≼PrimT x ⟹ ∃y. S = PrimT y"*
apply *(ind_cases "G⊢S≼T")*
by *auto*

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

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

lemma *widen_Iface: "G⊢Iface I≼T ⟹ T=Class Object ∨ (∃J. T=Iface J)"*
apply *(ind_cases "G⊢S≼T")*
by *auto*

lemma *widen_Iface2: "G⊢S≼ Iface J ⟹ S = NT ∨ (∃I. S = Iface I) ∨ (∃D. S = Class D)"*
apply *(ind_cases "G⊢S≼T")*
by *auto*

lemma *widen_Iface_Iface: "G⊢Iface I≼ Iface J ⟹ G⊢I≼I J"*
apply *(ind_cases "G⊢S≼T")*
by *auto*

lemma *widen_Iface_Iface_eq [simp]: "G⊢Iface I≼ Iface J = G⊢I≼I J"*
apply *(rule iffI)*

apply *(erule widen_Iface_Iface)*
apply *(erule widen.subint)*
done

lemma *widen_Class: "G⊢Class C≼T ⟹ (∃D. T=Class D) ∨ (∃I. T=Iface I)"*
apply *(ind_cases "G⊢S≼T")*
by *auto*

lemma *widen_Class2: "G⊢S≼ Class C ⟹ C = Object ∨ S = NT ∨ (∃D. S = Class D)"*
apply *(ind_cases "G⊢S≼T")*
by *auto*

lemma *widen_Class_Class: "G⊢Class C≼ Class cm ⟹ G⊢C≼C cm"*
apply *(ind_cases "G⊢S≼T")*
by *auto*

lemma *widen_Class_Class_eq [simp]: "G⊢Class C≼ Class cm = G⊢C≼C cm"*
apply *(rule iffI)*
apply *(erule widen_Class_Class)*
apply *(erule widen.subcls)*
done

lemma *widen_Class_Iface: "G⊢Class C≼ Iface I ⟹ G⊢C⇝I"*
apply *(ind_cases "G⊢S≼T")*
by *auto*

lemma *widen_Class_Iface_eq [simp]: "G⊢Class C≼ Iface I = G⊢C⇝I"*
apply *(rule iffI)*
apply *(erule widen_Class_Iface)*
apply *(erule widen.implmt)*
done

lemma *widen_Array: "G⊢S.[]≼T ⟹ T=Class Object ∨ (∃T'. T=T'.[] ∧ G⊢S≼T')"*
apply *(ind_cases "G⊢S≼T")*
by *auto*

lemma *widen_Array2: "G⊢S≼T.[] ⟹ S = NT ∨ (∃S'. S=S'.[] ∧ G⊢S'≼T)"*
apply *(ind_cases "G⊢S≼T")*
by *auto*

lemma *widen_ArrayPrimT: "G⊢PrimT t.[]≼T ⟹ T=Class Object ∨ T=PrimT t.[]"*
apply *(ind_cases "G⊢S≼T")*
by *auto*

```
lemma widen_ArrayRefT:
  "G⊢RefT t.[]≼T ⟹ T=Class Object ∨ (∃s. T=RefT s.[] ∧ G⊢RefT t≼RefT
s)"
apply (ind_cases "G⊢S≼T")
by auto

lemma widen_ArrayRefT_ArrayRefT_eq [simp]:
  "G⊢RefT T.[]≼RefT T'.[] = G⊢RefT T≼RefT T'"
apply (rule iffI)
apply (drule widen_ArrayRefT)
apply simp
apply (erule widen.array)
done

lemma widen_Array_Array: "G⊢T.[]≼T'.[] ⟹ G⊢T≼T'"
apply (drule widen_Array)
apply auto
done

lemma widen_NT2: "G⊢S≼NT ⟹ S = NT"
apply (ind_cases "G⊢S≼T")
by auto

lemma widen_trans_lemma [rule_format (no_asm)]:
  "⟦G⊢S≼U; ∀C. is_class G C ⟶ G⊢C≼C Object⟧ ⟹ ∀T. G⊢U≼T ⟶ G⊢S≼T"
apply (erule widen.induct)
apply        safe
prefer      5 apply (drule widen_RefT) apply clarsimp
apply      (frule_tac [1] widen_Iface)
apply      (frule_tac [2] widen_Class)
apply      (frule_tac [3] widen_Class)
apply      (frule_tac [4] widen_Iface)
apply      (frule_tac [5] widen_Class)
apply      (frule_tac [6] widen_Array)
apply      safe
apply           (rule widen.int_obj)
prefer          6 apply (drule implmt_is_class) apply simp
apply (tactic "ALLGOALS (etac thin_rl)")
prefer         6 apply simp
apply          (rule_tac [9] widen.arr_obj)
apply         (rotate_tac [9] -1)
apply         (frule_tac [9] widen_RefT)
apply         (auto elim!: rtrancl_trans subcls_implmt implmt_subint2)
done

lemma ws_widen_trans: "⟦G⊢S≼U; G⊢U≼T; ws_prog G⟧ ⟹ G⊢S≼T"
by (auto intro: widen_trans_lemma subcls_ObjectI)

lemma widen_antisym_lemma [rule_format (no_asm)]: "⟦G⊢S≼T;
```

```
 ∀I J. G⊢I≼I J ∧ G⊢J≼I I ⟶ I = J;
 ∀C D. G⊢C≼C D ∧ G⊢D≼C C ⟶ C = D;
 ∀I  . G⊢Object⇝I        ⟶ False⟧ ⟹ G⊢T≼S ⟶ S = T"
apply (erule widen.induct)
apply (auto dest: widen_Iface widen_NT2 widen_Class)
done

lemmas subint_antisym =
       subint1_acyclic [THEN acyclic_impl_antisym_rtrancl, standard]
lemmas subcls_antisym =
       subcls1_acyclic [THEN acyclic_impl_antisym_rtrancl, standard]

lemma widen_antisym: "⟦G⊢S≼T; G⊢T≼S; ws_prog G⟧ ⟹ S=T"
by (fast elim: widen_antisym_lemma subint_antisym [THEN antisymD]
                                   subcls_antisym [THEN antisymD])

lemma widen_ObjectD [dest!]: "G⊢Class Object≼T ⟹ T=Class Object"
apply (frule widen_Class)
apply (fast dest: widen_Class_Class widen_Class_Iface)
done

constdefs
  widens :: "prog ⇒ [ty list, ty list] ⇒ bool" ("_⊢_[≼]_" [71,71,71]
70)
 "G⊢Ts[≼]Ts' ≡ list_all2 (λT T'. G⊢T≼T') Ts Ts'"

lemma widens_Nil [simp]: "G⊢[][≼][]"
apply (unfold widens_def)
apply auto
done

lemma widens_Cons [simp]: "G⊢(S#Ss)[≼](T#Ts) = (G⊢S≼T ∧ G⊢Ss[≼]Ts)"
apply (unfold widens_def)
apply auto
done
```

25 narrowing relation

inductive *"narrow G"* **intros**

```
  subcls: "G⊢C≼C D              ⟹ G⊢     Class D≻Class C"
  implmt: "¬G⊢C⇝I               ⟹ G⊢     Class C≻Iface I"
  obj_arr:                        "G⊢Class Object≻T.[]"
  int_cls:                        "G⊢     Iface I≻Class C"
  subint: "imethds G I hidings imethds G J entails
          (λ(md, mh   ) (md',mh').  G⊢mrt mh≼mrt mh') ⟹
          ¬G⊢I≼I J              ⟹ G⊢     Iface I≻Iface J"
  array:  "G⊢RefT S≻RefT T      ⟹ G⊢   RefT S.[]≻RefT T.[]"
```

lemma *narrow_RefT: "G⊢RefT R≻T ⟹ ∃t. T=RefT t"*
apply *(ind_cases "G⊢S≻T")*
by *auto*

lemma *narrow_RefT2: "G⊢S≻RefT R ⟹ ∃t. S=RefT t"*
apply *(ind_cases "G⊢S≻T")*
by *auto*

lemma *narrow_PrimT: "G⊢PrimT pt≻T ⟹ ∃t. T=PrimT t"*
apply *(ind_cases "G⊢S≻T")*
by *auto*

lemma *narrow_PrimT2: "G⊢S≻PrimT pt ⟹*
∃t. S=PrimT t ∧ G⊢PrimT t≼PrimT pt"
apply *(ind_cases "G⊢S≻T")*
by *auto*

26 casting relation

inductive *"cast G"* **intros**

widen: "G⊢S≼T ⟹ G⊢S≼? T"
narrow: "G⊢S≻T ⟹ G⊢S≼? T"

lemma *cast_RefT: "G⊢RefT R≼? T ⟹ ∃t. T=RefT t"*
apply *(ind_cases "G⊢S≼? T")*
by *(auto dest: widen_RefT narrow_RefT)*

lemma *cast_RefT2: "G⊢S≼? RefT R ⟹ ∃t. S=RefT t"*
apply *(ind_cases "G⊢S≼? T")*
by *(auto dest: widen_RefT2 narrow_RefT2)*

lemma *cast_PrimT: "G⊢PrimT pt≼? T ⟹ ∃t. T=PrimT t"*
apply *(ind_cases "G⊢S≼? T")*
by *(auto dest: widen_PrimT narrow_PrimT)*

lemma *cast_PrimT2: "G⊢S≼? PrimT pt ⟹ ∃t. S=PrimT t ∧ G⊢PrimT t≼PrimT pt"*
apply *(ind_cases "G⊢S≼? T")*
by *(auto dest: widen_PrimT2 narrow_PrimT2)*

end

theory *WellType = TypeRel:*

types *lenv*
= "(lname, ty) table"

env
= "prog × lenv"

syntax
prg :: "env ⇒ prog"
lcl :: "env ⇒ lenv"

translations
"lenv" <= (type) "(lname, ty) table"
"env" <= (type) "prog × lenv"
"prg" => "fst"
"lcl" => "snd"

27 maximally specific methods

types
emhead = "ref_ty × mhead"
consts
cmheads :: "prog ⇒ tname ⇒ sig ⇒ emhead set"
mheads :: "prog ⇒ ref_ty ⇒ sig ⇒ emhead set"
defs
cmheads_def:
"cmheads G C ≡ λsig. (λ(C,(h,b)). (ClassT C,h)) ‘ o2s (cmethd G C sig)"
primrec
"mheads G NullT = (λsig. {})"
"mheads G (IfaceT I) = (λsig. (λ(I,h).(IfaceT I,h)) ‘ imethds G I sig
∪
cmheads G Object sig)"
"mheads G (ClassT C) = cmheads G C"
"mheads G (ArrayT T) = cmheads G Object"

constdefs

appl_methds :: "prog ⇒ ref_ty ⇒ sig ⇒ (emhead × ty list) set"
"appl_methds G rt ≡ λ(mn, pTs). {(mh,pTs’) |mh pTs’.
mh ∈ mheads G rt (mn, pTs’) ∧ G⊢pTs[≼]pTs’}"

more_spec :: "prog ⇒ emhead × ty list ⇒ emhead × ty list ⇒
bool"

```
 "more_spec G ≡ λ(mh,pTs). λ(mh',pTs'). G⊢pTs[≼]pTs'"


   max_spec     :: "prog ⇒  ref_ty ⇒ sig ⇒ (emhead × ty list)   set"

 " max_spec G rt sig ≡{m. m ∈appl_methds G rt sig ∧
                       (∀m'∈appl_methds G rt sig. more_spec G m' m ⟶
m'=m)}"


lemma max_spec2appl_meths: "x ∈ max_spec G T sig ⟹ x ∈ appl_methds
G T sig"
by (auto simp: max_spec_def)

lemma appl_methsD: "(mh,pTs')∈appl_methds G T (mn, pTs) ⟹
   mh ∈ mheads G T (mn, pTs') ∧ G⊢pTs[≼]pTs'"
by (auto simp: appl_methds_def)

lemma max_spec2mheads: "max_spec G rt (mn, pTs) = insert (mh, pTs') A
⟹
   mh ∈ mheads G rt (mn, pTs') ∧ G⊢pTs[≼]pTs'"
apply (auto dest: equalityD2 subsetD max_spec2appl_meths appl_methsD)
done


constdefs
  empty_dt :: "dyn_ty"
 "empty_dt ≡ λa. None"

  invmode :: "modi ⇒ expr ⇒ inv_mode"
 "invmode m e ≡ if static m then Static else if e=Super then SuperM else
IntVir"

lemma invmode_nonstatic: "invmode False (Acc (LVar e)) = IntVir"
apply (unfold invmode_def)
apply (simp (no_asm))
done
declare invmode_nonstatic [simp]

lemma invmode_Static_eq: "(invmode m e = Static) = m"
apply (unfold invmode_def)
apply (simp (no_asm))
done
declare invmode_Static_eq [simp]

lemma invmode_IntVir_eq: "(invmode m e = IntVir) = (¬m ∧ e≠Super)"
apply (unfold invmode_def)
apply (simp (no_asm))
```

done

lemma *Null_staticD: "a'=Null ⟶ m ⟹ invmode m e = IntVir ⟶ a' ≠ Null"*
apply *(clarsimp simp add: invmode_IntVir_eq)*
done

types *tys = "ty + ty list"*
translations
"tys" <= (type) "ty + ty list"

consts
wt :: "(env × dyn_ty × term × tys) set"

syntax
wt :: "env ⇒ dyn_ty ⇒ [term,tys] ⇒ bool" ("_,_|=_::_" [51,51,51,51] 50)
wt_stmt :: "env ⇒ dyn_ty ⇒ stmt ⇒ bool" ("_,_|=_:<>" [51,51,51] 50)
ty_expr :: "env ⇒ dyn_ty ⇒ [expr ,ty] ⇒ bool" ("_,_|=_:-_" [51,51,51,51] 50)
ty_var :: "env ⇒ dyn_ty ⇒ [var ,ty] ⇒ bool" ("_,_|=_:=_" [51,51,51,51] 50)
ty_exprs:: "env ⇒ dyn_ty ⇒ [expr list, ty list] ⇒ bool" ("_,_|=_:#_" [51,51,51,51] 50)

syntax *(xsymbols)*
wt :: "env ⇒ dyn_ty ⇒ [term,tys] ⇒ bool" ("_,_⊨_::_" [51,51,51,51] 50)
wt_stmt :: "env ⇒ dyn_ty ⇒ stmt ⇒ bool" ("_,_⊨_::√" [51,51,51] 50)
ty_expr :: "env ⇒ dyn_ty ⇒ [expr ,ty] ⇒ bool" ("_,_⊨_::-_" [51,51,51,51] 50)
ty_var :: "env ⇒ dyn_ty ⇒ [var ,ty] ⇒ bool" ("_,_⊨_::=_" [51,51,51,51] 50)
ty_exprs:: "env ⇒ dyn_ty ⇒ [expr list, ty list] ⇒ bool"("_,_⊨_::�doteq_" [51,51,51,51] 50)

translations
"E,dt⊨t::T" == "(E,dt,t,T) ∈ wt"
"E,dt⊨s::√" == "E,dt⊨In1r s::Inl (PrimT Void)"
"E,dt⊨e::-T" == "E,dt⊨In1l e::Inl T"
"E,dt⊨e::=T" == "E,dt⊨In2 e::Inl T"
"E,dt⊨e::≐T" == "E,dt⊨In3 e::Inr T"

syntax

```
  wt_      :: "env ⇒ [term,tys] ⇒ bool" ("_|-_::_" [51,51,51] 50)
  wt_stmt_ :: "env ⇒  stmt      ⇒ bool" ("_|-_:<>" [51,51   ] 50)
  ty_expr_ :: "env ⇒ [expr ,ty ] ⇒ bool" ("_|-_:-_" [51,51,51] 50)
  ty_var_  :: "env ⇒ [var  ,ty ] ⇒ bool" ("_|-_:=_" [51,51,51] 50)
  ty_exprs_:: "env ⇒ [expr list,
                      ty   list] ⇒ bool" ("_|-_:#_" [51,51,51] 50)
```

syntax *(xsymbols)*

```
  wt_      :: "env ⇒ [term,tys] ⇒ bool" ("_⊢_::_"  [51,51,51] 50)
  wt_stmt_ ::  "env ⇒  stmt      ⇒ bool" ("_⊢_::√"  [51,51   ] 50)
  ty_expr_ :: "env ⇒ [expr ,ty ] ⇒ bool" ("_⊢_::-_" [51,51,51] 50)
  ty_var_  :: "env ⇒ [var  ,ty ] ⇒ bool" ("_⊢_::=_" [51,51,51] 50)
  ty_exprs_ :: "env ⇒ [expr list,
                       ty   list] ⇒ bool" ("_⊢_::�doteq_" [51,51,51] 50)
```

translations

```
        "E⊢t:: T" == "E,empty_dt⊨t:: T"
        "E⊢s::√" == "E,empty_dt⊨s::√"
        "E⊢e::-T" == "E,empty_dt⊨e::-T"
        "E⊢e::=T" == "E,empty_dt⊨e::=T"
        "E⊢e::≐T" == "E,empty_dt⊨e::≐T"
```

inductive *wt* **intros**

```
  Skip:                                          "E,dt⊨Skip::√"

  Expr: "⟦E,dt⊨e::-T⟧ ⟹
                                                  E,dt⊨Expr e::√"

  Comp: "⟦E,dt⊨c1::√;
          E,dt⊨c2::√⟧ ⟹
                                                  E,dt⊨c1;; c2::√"


  If:   "⟦E,dt⊨e::-PrimT Boolean;
          E,dt⊨c1::√;
          E,dt⊨c2::√⟧ ⟹
                                                  E,dt⊨If(e) c1 Else c2::√"


  Loop: "⟦E,dt⊨e::-PrimT Boolean;
          E,dt⊨c::√⟧ ⟹
                                                  E,dt⊨While(e) c::√"


  Throw: "⟦E,dt⊨e::-Class tn;
```

```
        prg E⊢tn⪯C SXcpt Throwable⟧ ⟹
                                        E,dt⊨Throw e::√"

  Try:  "⟦E,dt⊨c1::√; prg E⊢tn⪯C SXcpt Throwable;
        lcl E (EName vn)=None; (prg E,lcl E(EName vn↦Class tn)),dt⊨c2::√⟧
⟹
                                        E,dt⊨Try c1 Catch(tn vn) c2::√"

  Fin:  "⟦E,dt⊨c1::√; E,dt⊨c2::√⟧ ⟹
                                        E,dt⊨c1 Finally c2::√"

  Init: "⟦is_class (prg E) C⟧ ⟹
                                        E,dt⊨init C::√"

  NewC: "⟦is_class (prg E) C⟧ ⟹
                                        E,dt⊨NewC C::-Class C"

  NewA: "⟦is_type (prg E) T;
        E,dt⊨i::-PrimT Integer⟧ ⟹
                                        E,dt⊨New T[i]::-T.[]"

  Cast: "⟦E,dt⊨e::-T; is_type (prg E) T';
        prg E⊢T⪯? T'⟧ ⟹
                                        E,dt⊨Cast T' e::-T'"

  Inst: "⟦E,dt⊨e::-RefT T;
        prg E⊢RefT T⪯? RefT T'⟧ ⟹
                                        E,dt⊨e InstOf T'::-PrimT Boolean"

  Lit:  "⟦typeof dt x = Some T⟧ ⟹
                                        E,dt⊨Lit x::-T"

  Super: "⟦lcl E This = Some (Class C); C ≠ Object;
        class (prg E) C = Some (D, rest)⟧ ⟹
                                        E,dt⊨Super::-Class D"

  Acc:  "⟦E,dt⊨va::=T⟧ ⟹
                                        E,dt⊨Acc va::-T"
```

```
Ass:  "⟦E,dt⊨va::=T; va ≠ LVar This;
        E,dt⊨v ::-T';
        prg E⊢T'⪯T⟧ ⟹
                                        E,dt⊨va:=v::-T'"


Cond: "⟦E,dt⊨e0::-PrimT Boolean;
        E,dt⊨e1::-T1; E,dt⊨e2::-T2;
        prg E⊢T1⪯T2 ∧ T = T2  ∨  prg E⊢T2⪯T1 ∧ T = T1⟧ ⟹
                                        E,dt⊨e0 ? e1 : e2::-T"


Call: "⟦E,dt⊨e::-RefT t;
        E,dt⊨ps::≐pTs;
        max_spec (prg E) t (mn, pTs) = {((md,(m,pns,rT)),pTs')}⟧ ⟹
                                E,dt⊨{t,md,invmode m e}e..mn({pTs'}ps)::-rT"

Methd: "⟦is_class (prg E) C;
        cmethd (prg E) C sig = Some (md,mh,lvars,blk,res);
        E,dt⊨Body md blk res::-T⟧ ⟹
                                        E,dt⊨Methd C sig::-T"

Body: "⟦is_class (prg E) D;
        E,dt⊨blk::√;
        E,dt⊨res::-T⟧ ⟹
                                        E,dt⊨Body D blk res::-T"



LVar: "⟦lcl E vn = Some T; is_type (prg E) T⟧ ⟹
                                        E,dt⊨LVar vn::=T"

FVar: "⟦E,dt⊨e::-Class C;
        cfield (prg E) C fn = Some (fd,(m,fT))⟧ ⟹
                                        E,dt⊨{fd,static m}e..fn::=fT"

AVar: "⟦E,dt⊨e::-T.[];
        E,dt⊨i::-PrimT Integer⟧ ⟹
                                        E,dt⊨e.[i]::=T"



Nil:                                  "E,dt⊨[]::≐[]"
```

```
  Cons: "⟦E,dt⊨e ::-T;
          E,dt⊨es::≐Ts⟧ ⟹
                                              E,dt⊨e#es::≐T#Ts"
```

declare *not_None_eq [simp del]*
declare *split_if [split del] split_if_asm [split del]*
declare *split_paired_All [simp del] split_paired_Ex [simp del]*
ML_setup *{**
simpset_ref() := simpset() delloop "split_all_tac"
**}*
inductive_cases *wt_stmt_cases: "E,dt⊨c::√"*
inductive_cases *wt_elim_cases:*

```
        "E,dt⊨In2  (LVar vn)               ::T"
        "E,dt⊨In2  ({fd,s}e..fn)           ::T"
        "E,dt⊨In2  (e.[i])                 ::T"
        "E,dt⊨In1l (NewC C)                ::T"
        "E,dt⊨In1l (New T'[i])             ::T"
        "E,dt⊨In1l (Cast T' e)             ::T"
        "E,dt⊨In1l (e InstOf T')           ::T"
        "E,dt⊨In1l (Lit x)                 ::T"
        "E,dt⊨In1l (Super)                 ::T"
        "E,dt⊨In1l (Acc va)                ::T"
        "E,dt⊨In1l (Ass va v)              ::T"
        "E,dt⊨In1l (e0 ? e1 : e2)          ::T"
        "E,dt⊨In1l ({t,md,mode}e..mn({pT'}p))::T"
        "E,dt⊨In1l (Methd C sig)           ::T"
        "E,dt⊨In1l (Body D blk res)        ::T"
        "E,dt⊨In3  ([])                    ::Ts"
        "E,dt⊨In3  (e#es)                  ::Ts"
        "E,dt⊨In1r  Skip                   ::x"
        "E,dt⊨In1r (Expr e)                ::x"
        "E,dt⊨In1r (c1;; c2)               ::x"
        "E,dt⊨In1r (If(e) c1 Else c2)      ::x"
        "E,dt⊨In1r (While(e) c)            ::x"
        "E,dt⊨In1r (Throw e)               ::x"
        "E,dt⊨In1r (Try c1 Catch(tn vn) c2)::x"
        "E,dt⊨In1r (c1 Finally c2)         ::x"
        "E,dt⊨In1r (init C)                ::x"
```

declare *not_None_eq [simp]*
declare *split_if [split] split_if_asm [split]*
declare *split_paired_All [simp] split_paired_Ex [simp]*
ML_setup *{**
simpset_ref() := simpset() addloop ("split_all_tac", split_all_tac)
**}*

lemma *wt_Methd_is_methd: "(G,L)⊢In1l (Methd C sig)::T ⟹ is_methd G C sig"*

```
apply (erule_tac wt_elim_cases)
apply clarsimp
apply (erule is_methdI, assumption)
done

lemma wt_Call: "⟦E,dt⊨e::-RefT t; E,dt⊨ps::�doteq pTs;
       max_spec (prg E) t (mn, pTs) = {((md,(m,pns,rT)),pTs')};
       mode = invmode m e⟧ ⟹ E,dt⊨{t,md,mode}e..mn({pTs'}ps)::-rT"
apply (rotate_tac -1)
apply (erule ssubst)
apply (erule wt.Call)
apply auto
done

lemma wt_init [iff]: "E,dt⊨init C::√ = is_class (prg E) C"
by (auto elim: wt_elim_cases intro: "wt.Init")

declare wt.Skip [iff]

lemma wt_StatRef: "isrtype (fst E) rt ⟹ E⊢StatRef rt::-RefT rt"
apply (rule wt.Cast)
apply  (rule wt.Lit)
apply   (simp (no_asm))
apply  (simp (no_asm_simp))
apply (rule cast.widen)
apply (simp (no_asm))
done

lemma wt_Inj_elim: "⋀E. E,dt⊨t::U ⟹ case t of In1 ec ⇒ (case ec
of Inl e ⇒ ∃T. U=Inl T
                                          |      Inr s ⇒ U=Inl (PrimT
Void))
   | In2 e ⇒ (∃T. U=Inl T) | In3 e ⇒ (∃T. U=Inr T)"
apply (simp (no_asm_simp) only: split_tupled_all)
apply (erule wt.induct)
apply auto
done

ML {*
fun wt_fun name inj rhs =
let
  val lhs = "E,dt⊨" ^ inj ^ " t::U"
  val () = qed_goal name (the_context()) (lhs ^ " = (" ^ rhs ^ ")")
        (K [Auto_tac, ALLGOALS (ftac (thm "wt_Inj_elim")) THEN Auto_tac])
  fun is_Inj (Const (inj,_) $ _) = true
    | is_Inj _                   = false
  fun pred (t as (_ $ (Const ("Pair",_) $
     _ $ (Const ("Pair", _) $ _ $ (Const ("Pair", _) $ _ $
       x))) $ _ )) = is_Inj x
```

```
in
  make_simproc name lhs pred (thm name)
end

val wt_expr_proc  = wt_fun "wt_expr_eq"  "In1l" "∃T.  U=Inl T  ∧ E,dt⊨t::-T"
val wt_var_proc   = wt_fun "wt_var_eq"   "In2"  "∃T.  U=Inl T  ∧ E,dt⊨t::=T"
val wt_exprs_proc = wt_fun "wt_exprs_eq" "In3"  "∃Ts. U=Inr Ts ∧ E,dt⊨t::≐Ts"
val wt_stmt_proc  = wt_fun "wt_stmt_eq"  "In1r" "U=Inl(PrimT Void)∧E,dt⊨t::√"
*}

ML {*
Addsimprocs [wt_expr_proc,wt_var_proc,wt_exprs_proc,wt_stmt_proc]
*}

lemma Inj_eq_lemma [simp]:
  "(∀T. (∃T'. T = Inj T' ∧ P T') ⟶ Q T) = (∀T'. P T' ⟶ Q (Inj T'))"
by auto


lemma single_valued_tys_lemma [rule_format (no_asm)]: "∀S T. G⊢S≼T
⟶ G⊢T≼S ⟶ S = T ⟹ E,dt⊨t::T ⟹
  G = fst E ⟶ (∀T'. E,dt⊨t::T' ⟶ T  = T')"
apply (case_tac "E", case_tac "a", erule wt.induct)
apply (safe del: disjE)
apply (simp_all (no_asm_use) split del: split_if_asm)
apply (safe del: disjE)

apply (tactic {* ALLGOALS (fn i => if i = 9 then EVERY'[thin_tac "?E,dt⊨e0::-PrimT
Boolean", thin_tac "?E,dt⊨e1::-?T1", thin_tac "?E,dt⊨e2::-?T2"] i else
thin_tac "All ?P" i) *})

apply (tactic {*ALLGOALS (eresolve_tac (thms "wt_elim_cases"))*})
apply (simp_all (no_asm_use) split del: split_if_asm)
apply (erule_tac [10] V = "All ?P" in thin_rl)
apply ((blast del: equalityCE dest: sym [THEN trans])+)
done


lemma single_valued_tys:
"ws_prog (fst E) ⟹ single_valued {(t,T). E,dt⊨t::T}"
apply (unfold single_valued_def)
apply clarsimp
apply (rule single_valued_tys_lemma)
apply (auto intro!: widen_antisym)
done

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

```
apply           auto
done


lemma typeof_is_type [rule_format (no_asm)]: "(∀a. v ≠ Addr a) ⟶
(∃T. typeof dt v = Some T ∧ is_type G T)"
apply (rule val.induct)
prefer 5
apply     fast
apply  (simp_all (no_asm))
done

end


theory WellForm = WellType:
```

28 well-formed field declarations

```
constdefs
  wf_fdecl :: "prog ⇒ fdecl ⇒ bool"
 "wf_fdecl G ≡ λ(fn,(m,ft)). is_type G ft"

lemma wf_fdecl_def2: "⋀fd. wf_fdecl G fd = is_type G (snd (snd fd))"
apply (unfold wf_fdecl_def)
apply simp
done
```

29 well-formed method declarations

```
constdefs

  wf_mhead :: "prog ⇒ sig ⇒ mhead ⇒ bool"
 "wf_mhead G ≡ λ(mn,pTs) (m,pns,rT). length pTs = length pns ∧
                                ( ∀T∈set pTs. is_type G T) ∧ is_type
G rT ∧
                                nodups pns"


  wf_mdecl :: "prog ⇒ tname ⇒ mdecl ⇒ bool"
 "wf_mdecl G C ≡ λ((mn,pTs),(m,pns,rT),lvars,blk,res).
                 wf_mhead G        (mn,pTs) (m,pns,rT) ∧ unique lvars
∧
                (C=Object ⟶ ¬static m) ∧ (∀(vn,T)∈set lvars. is_type
G T) ∧
                (∀pn∈set pns. table_of lvars pn = None) ∧
                (∃T. (G,table_of lvars(pns[↦]pTs) (+)
                        (if static m then empty else empty(()↦Class C)))⊢
```

```
                    Body C blk res::-T ∧ G⊢T⪯rT)"

lemma wf_mheadI: "⟦length pTs = length pns;
  ∀T∈set pTs. is_type G T; is_type G rT; nodups pns⟧ ⟹
  wf_mhead G (mn, pTs) (m, pns, rT)"
apply (unfold wf_mhead_def)
apply (simp (no_asm_simp))
done

lemma wf_mdeclI: "⟦(C = Object ⟶ ¬ static m);
  wf_mhead G (mn, pTs) (m, pns, rT); unique lvars;
  (∀pn∈set pns. table_of lvars pn = None);
  ∀(vn,T)∈set lvars. is_type G T;
  (G, table_of lvars(pns[↦]pTs) (+)
      (if static m then empty else empty(()↦Class C)))⊢Body C blk res::-T;

  G⊢T⪯rT⟧ ⟹
  wf_mdecl G C ((mn, pTs), (m, pns, rT), lvars, blk, res)"
apply (unfold wf_mdecl_def)
apply (auto split del: split_if split_if_asm)
done

lemma wf_mdeclD1:
"⋀sig. wf_mdecl G C (sig,(m,pns,rT),lvars,blk,res) ⟹
 wf_mhead G    sig (m,pns,rT) ∧ unique lvars ∧
  (∀pn∈set pns. table_of lvars pn = None) ∧ (∀(vn,T)∈set lvars. is_type
G T)"
apply (unfold wf_mdecl_def)
apply auto
done

lemma wf_mdecl_bodyD:
"⋀sig. wf_mdecl G C (sig,(m,pns,rT),lvars,blk,res) ⟹
 (∃T. (G,table_of lvars(pns[↦](snd sig)) (+)
  (if static m then empty else empty(()↦Class C)))⊢Body C blk res::-T∧
G⊢T⪯rT)"
apply (unfold wf_mdecl_def)
apply auto
done

lemma static_Object_methodsE [elim!]:
 "⋀sig. wf_mdecl G Object (sig, (True, pns, T), b) ⟹ R"
apply (unfold wf_mdecl_def)
apply auto
done

lemma rT_is_type: "⋀sig. wf_mhead G sig (m,pns,rT) ⟹ is_type G rT"
apply (unfold wf_mhead_def)
apply auto
```

done

30 well-formed interface declarations

constdefs

```
  wf_idecl :: "prog ⇒ idecl ⇒ bool"
 "wf_idecl G ≡ λ(I,(si,ms)). ws_idecl G I si ∧
                ¬is_class G I ∧
                (∀(sig,mh)∈set ms. wf_mhead G sig mh ∧ ¬static (fst
mh)) ∧
                unique ms ∧
                (o2s ∘ table_of ms hidings Un_tables((λJ.(imethds G J))‘set
si)
                 entails (λmh (md,mh'). G⊢mrt mh⪯mrt mh'))"
```

lemma *wf_idecl_mhead:* "⋀m. ⟦wf_idecl G (I, is, ms); (s,mh)∈set ms⟧ ⟹

```
  wf_mhead G s mh ∧ ¬static (fst mh)"
```

apply *(unfold wf_idecl_def)*
apply *auto*
done

lemma *wf_idecl_hidings:*

```
"wf_idecl G (I, is, ms) ⟹ (λs. o2s (table_of ms s)) hidings
  Un_tables ((λJ. imethds G J) ‘ set is)
  entails λmh (md,mh'). G⊢mrt mh⪯mrt mh'"
```

apply *(unfold wf_idecl_def o_def)*
apply *simp*
done

lemma *wf_idecl_supD:*

```
"⟦wf_idecl G (I, is, ms); J ∈ set is⟧ ⟹ is_iface G J ∧ (J, I) ∉ (subint1
G)^+"
```

apply *(unfold wf_idecl_def ws_idecl_def)*
apply *auto*
done

31 well-formed class declarations

constdefs

```
  wf_cdecl :: "prog ⇒ cdecl ⇒ bool"
 "wf_cdecl G ≡ λ(C,(sc,si,fs,ms,init)).
                ¬is_iface G C ∧
                (∀I∈set si. is_iface G I ∧
                     (∀s. ∀(md', mh'   ) ∈ imethds G I s.
                         (∃(md ,(mh ,b)) ∈ cmethd  G C s: G⊢mrt mh⪯mrt
mh' ∧
                                                          ¬static (fst
```

```
mh)))) ∧
                (∀f∈set fs. wf_fdecl G   f) ∧ unique fs ∧
                (∀m∈set ms. wf_mdecl G C m) ∧ unique ms ∧
                (G,empty)⊢init::√ ∧ ws_cdecl G C sc ∧
                (C ≠ Object ⟶ (table_of ms hiding cmethd G sc entails
                                  (λ(mh,b) (md',(mh',b')). G⊢mrt mh⪯mrt
mh' ∧
                                                 static (fst mh') = static (fst
mh))))"

lemma wf_cdecl_unique:
"wf_cdecl G (C,sc,si,fs,ms,ini) ⟹ unique fs ∧ unique ms"
apply (unfold wf_cdecl_def)
apply auto
done

lemma wf_cdecl_fdecl:
"⟦wf_cdecl G (C,sc,si,fs,ms,ini); f∈set fs⟧ ⟹ wf_fdecl G f"
apply (unfold wf_cdecl_def)
apply auto
done

lemma wf_cdecl_mdecl:
"⟦wf_cdecl G (C,sc,si,fs,ms,ini); m∈set ms⟧ ⟹ wf_mdecl G C m"
apply (unfold wf_cdecl_def)
apply auto
done

lemma wf_cdecl_impD:
"⟦wf_cdecl G (C,sc,si,fs,ms,ini); I∈set si⟧ ⟹ is_iface G I ∧
    (∀s. ∀(md',mh') ∈ imethds G I s.
     (∃(md,(mh,b))∈cmethd G C s: G⊢mrt mh⪯mrt mh' ∧ ¬static (fst mh)))"
apply (unfold wf_cdecl_def)
apply auto
done

lemma wf_cdecl_supD:
"⟦wf_cdecl G (C,sc,si,fs,ms,ini); C ≠ Object⟧ ⟹
  is_class G sc ∧ (sc,C) ∉ (subcls1 G)^+ ∧ (table_of ms hiding cmethd
G sc
  entails (λ(mh,b) (md',(mh',b')). G⊢mrt mh⪯mrt mh' ∧ fst mh' = fst
mh))"
apply (unfold wf_cdecl_def ws_cdecl_def)
apply auto
done

lemma wf_cdecl_wt_init:
 "wf_cdecl G (C, sco, si, fs, ms, ini) ⟹ (G, empty)⊢ini::√"
apply (unfold wf_cdecl_def)
```

apply *auto*
done

32 well-formed programs

constdefs
```
  wf_prog  :: "prog ⇒ bool"
 "wf_prog G ≡ let is = fst G; cs = snd G in
                        ObjectC ∈ set cs  ∧ (∀xn. SXcptC xn ∈ set cs)
∧
                (∀i∈set is. wf_idecl G i) ∧ unique is ∧
                (∀c∈set cs. wf_cdecl G c) ∧ unique cs"
```

lemma *wf_prog_idecl: "⟦iface G I = Some i; wf_prog G⟧ ⟹ wf_idecl G (I,i)"*
apply *(unfold wf_prog_def Let_def)*
apply *simp*
apply *(fast dest: map_of_SomeD)*
done

lemma *wf_prog_cdecl: "⟦class G C = Some c; wf_prog G⟧ ⟹ wf_cdecl G (C,c)"*
apply *(unfold wf_prog_def Let_def)*
apply *simp*
apply *(fast dest: map_of_SomeD)*
done

lemma *wf_ws_prog [elim!,simp]: "wf_prog G ⟹ ws_prog G"*
apply *(unfold wf_prog_def Let_def)*
apply *(rule ws_progI)*
apply *(simp_all (no_asm))*
apply *(fast dest!: wf_idecl_supD wf_cdecl_supD)+*
done

lemma *class_Object [simp]:*
"wf_prog G ⟹ class G Object = Some (arbitrary,[],[],Object_mdecls,Skip)"
apply *(unfold wf_prog_def Let_def ObjectC_def)*
apply *(fast dest!: map_of_SomeI)*
done

lemma *class_SXcpt [simp]:*
```
        "wf_prog G ⟹ class G (SXcpt xn) = Some (if xn = Throwable
  then Object else SXcpt Throwable, [], [], SXcpt_mdecls, Skip)"
```
apply *(unfold wf_prog_def Let_def SXcptC_def)*
apply *(fast dest!: map_of_SomeI)*
done

lemma *wf_ObjectC [simp]:*
```
        "wf_cdecl G ObjectC = (¬is_iface G Object ∧ Ball (set Object_mdecls)
```

```
  (wf_mdecl G Object) ∧ unique Object_mdecls)"
apply (unfold wf_cdecl_def ws_cdecl_def ObjectC_def)
apply (simp (no_asm))
done

lemma Object_is_class [simp,elim!]: "wf_prog G ⟹ is_class G Object"
apply (simp (no_asm_simp))
done

lemma SXcpt_is_class [simp,elim!]: "wf_prog G ⟹ is_class G (SXcpt
xn)"
apply (simp (no_asm_simp))
done

lemma cmethd_Object: "wf_prog G ⟹ cmethd G Object =
  table_of (map (λ(s,m). (s, Object, m)) Object_mdecls)"
apply (subst cmethd_rec)
apply auto
done

lemma fields_Object [simp]: "wf_prog G ⟹ fields G Object = []"
by (force intro: fields_emptyI)

lemma cfield_Object [simp]:
 "wf_prog G ⟹ cfield G Object = empty"
apply (unfold cfield_def)
apply ((simp (no_asm_simp)))
done

lemma fields_Throwable [simp]: "wf_prog G ⟹ fields G (SXcpt Throwable)
= []"
by (force intro: fields_emptyI)

lemma fields_SXcpt [simp]: "wf_prog G ⟹ fields G (SXcpt xn) = []"
apply (case_tac "xn = Throwable")
apply  (simp (no_asm_simp))
by (force intro: fields_emptyI)

lemmas widen_trans = ws_widen_trans [OF _ _ wf_ws_prog, elim]
lemma widen_trans2 [elim]: "⟦G⊢U≼T; G⊢S≼U; wf_prog G⟧ ⟹ G⊢S≼T"
apply (erule (2) widen_trans)
done

lemma Xcpt_subcls_Throwable [simp]:
"wf_prog G ⟹ G⊢SXcpt xn≼C SXcpt Throwable"
apply (rule SXcpt_subcls_Throwable_lemma)
apply auto
done
```

```
lemma unique_fields: "⟦is_class G C; wf_prog G⟧ ⟹ unique (fields G
C)"
apply (erule ws_unique_fields)
apply  (erule wf_ws_prog)
apply safe
apply (erule (1) wf_prog_cdecl [THEN wf_cdecl_unique [THEN conjunct1]])
done

lemma fields_mono: "⟦table_of (fields G C) fn = Some f; G⊢D≼C C; is_class
G D; wf_prog G⟧
   ⟹ table_of (fields G D) fn = Some f"
apply (rule map_of_SomeI)
apply  (erule (1) unique_fields)
apply (erule (1) map_of_SomeD [THEN fields_mono_lemma])
apply (erule wf_ws_prog)
done

lemma fields_is_type [elim]:
"⋀m. ⟦table_of (fields G C) m = Some(f,T); wf_prog G; is_class G C⟧ ⟹

      is_type G T"
apply (frule wf_ws_prog)
apply (force dest: fields_defpl [THEN conjunct1]
                   wf_prog_cdecl [THEN wf_cdecl_fdecl]
             simp add: wf_fdecl_def2)
done

lemma imethds_wf_mhead [rule_format (no_asm)]:
"⟦(md,mh) ∈ imethds G I sig; wf_prog G; is_iface G I⟧ ⟹
  wf_mhead G sig mh ∧ ¬ static (fst mh)"
apply (frule wf_ws_prog)
apply (drule (2) imethds_defpl [THEN conjunct1])
apply clarify
apply (frule (1) wf_prog_idecl, erule wf_idecl_mhead, erule map_of_SomeD)
done

lemma cmethd_wf_mdecl:
 "⟦cmethd G C sig = Some (md,m); wf_prog G; class G C = Some y⟧ ⟹
  G⊢C≼C md ∧ is_class G md ∧ wf_mdecl G md (sig,m)"
apply (frule wf_ws_prog)
apply (drule (1) cmethd_defpl)
apply  fast
apply clarsimp
apply (frule (1) wf_prog_cdecl, erule wf_cdecl_mdecl, erule map_of_SomeD)
done

lemmas cmethd_rT_is_type = cmethd_wf_mdecl [THEN conjunct2, THEN conjunct2,
               THEN wf_mdeclD1 [THEN conjunct1], THEN rT_is_type]
```

```
lemma subcls_methd_: "⋀sig. ⟦G⊢C≼C C'; is_class G C'; wf_prog G⟧ ⟹

  ∀(md ,(m ,pns ,rT ),mb )∈cmethd G C' sig:
  ∃(md',(m',pns',rT'),mb')∈cmethd G C  sig: static m'=static m ∧ G⊢rT'≼rT"
apply (erule converse_rtrancl_induct)
apply  clarsimp
apply (drule subcls1D)
apply clarsimp
apply (subst cmethd_rec, assumption, erule wf_ws_prog)
apply (unfold override_def)
apply clarsimp
apply (drule (2) wf_prog_cdecl [THEN wf_cdecl_supD])
apply clarsimp
apply (drule (2) hiding_entailsD)
apply clarsimp
apply fast
done
lemmas subcls_methd = subcls_methd_ [THEN ospec]


ML {* bind_thm("bexI'",permute_prems 0 1 bexI) *}
ML {* bind_thm("ballE'",permute_prems 1 1 ballE) *}
lemma subint_widen_imethds: "⟦G⊢I≼I J; wf_prog G; is_iface G J⟧ ⟹

  ∀(md, m ,pns ,rT ) ∈ imethds G J sig.
  ∃(md',m',pns',rT') ∈ imethds G I sig. static m'=static m ∧ G⊢rT'≼rT"
apply (erule converse_rtrancl_induct)
apply  (clarsimp elim!: bexI')
apply (drule subint1D)
apply clarify
apply (erule ballE')
apply  fast
apply (erule_tac V = "?x ∈ imethds G J sig" in thin_rl)
apply clarsimp
apply (subst imethds_rec, assumption, erule wf_ws_prog)
apply (unfold overrides_def)
apply (drule (1) wf_prog_idecl)
apply (auto intro!: table_of_map_SomeI bexI'
            dest: wf_idecl_mhead
                  imethds_wf_mhead [OF _ _ wf_idecl_supD [THEN conjunct1]])
apply (auto dest!: wf_idecl_hidings [THEN hidings_entailsD])
done


lemma implmt1_methd:
 "⋀sig. ⟦G⊢C⇝1I; wf_prog G; (md, m, pn, rT) ∈ imethds G I sig⟧ ⟹
```

```
  ∃(md',(m',pn',rT'),mb)∈cmethd G C sig: static m'=static m ∧ G⊢rT'≼rT"
apply (drule implmt1D)
apply clarify
apply (drule (2) wf_prog_cdecl [THEN wf_cdecl_impD])
apply (frule (1) imethds_wf_mhead)
apply  simp
apply force
done


lemma implmt_methd [rule_format (no_asm)]:
"⟦wf_prog G; G⊢T''⇝I⟧ ⟹ is_iface G I ⟶
 (∀(md, (m ,pn ,rT ))    ∈imethds G I   sig.
  ∃(md',(m',pn',rT'),mb')∈cmethd  G T'' sig: static m'=static m ∧ G⊢rT'≼rT)"
apply (frule implmt_is_class)
apply (erule implmt.induct)
apply   safe
apply   (drule (2) implmt1_methd)
apply   fast
apply  (drule (1) subint_widen_imethds)
apply   simp
apply  (drule (1) bspec)
apply  clarify
apply  (drule (2) implmt1_methd)
apply  force
apply (frule subcls1D)
apply (drule (1) bspec)
apply clarify
apply (drule (3) r_into_rtrancl [THEN subcls_methd, OF _ implmt_is_class])
apply force
done

lemma mheadsD [rule_format (no_asm)]: "(md,mh) ∈ mheads G t sig ⟶
(∃C D b. t = ClassT C ∧ md = ClassT D ∧ cmethd G C sig = Some (D, mh,
b)) ∨ (∃I. t = IfaceT I ∧ (∃I'. (I',mh) ∈ imethds G I sig) ∨ (∃D b.
cmethd G Object sig = Some (D, mh, b))) ∨ (∃T D b. t = ArrayT T ∧ cmethd
G Object sig = Some (D, mh, b))"
apply (rule ref_ty_ty.induct [THEN conjunct1])
apply      (auto simp add: cmheads_def)
done

lemma class_mheadsD: "⋀sig mh.
 ⟦(md , m ,pn, rT )∈mheads G t sig; wf_prog G; class G C = Some y;
   if (∃T. t=ArrayT T) then C=Object else G⊢Class C≼RefT t; isrtype
G t⟧ ⟹
  ∃(md',(m',pn',rT'),mb)∈cmethd G C sig: static m'=static m ∧ G⊢rT'≼rT"
apply (drule mheadsD)
apply safe
apply    (tactic "ALLGOALS Clarsimp_tac")
```

```
apply   (force dest!: subcls_methd del: bexI)
apply  (drule (1) implmt_methd, simp, assumption, clarsimp)
apply (tactic {* dtac (permute_prems 0 ~1 (thm "subcls_methd")) 1 *})
apply    (auto intro!: subcls_ObjectI)
done

lemma wt_is_type: "E,dt⊨v::T ⟹ E=(G,L) ⟶ wf_prog G ⟶ dt=empty_dt
⟶
  (case T of Inl T ⇒ is_type G T | Inr Ts ⇒ Ball (set Ts) (is_type G))"
apply (unfold empty_dt_def)
apply (erule wt.induct)
apply (auto split del: split_if_asm simp del: snd_conv)
apply    (erule typeof_empty_is_type)
apply   (frule (1) wf_prog_cdecl [THEN wf_cdecl_supD], rotate_tac -1,

        force simp del: snd_conv, clarsimp)
apply  (drule max_spec2mheads [THEN conjunct1, THEN mheadsD])
apply  (drule_tac [2] cfield_fields)
apply  (auto elim: cmethd_rT_is_type
                  imethds_wf_mhead [THEN conjunct1, THEN rT_is_type]

            simp del: fst_conv snd_conv)
done
lemma ty_expr_is_type:
"⟦(G, L)⊢e::-T; wf_prog G⟧ ⟹ is_type G T"
by (auto dest!: wt_is_type)
lemma ty_var_is_type:
"⟦(G, L)⊢v::=T; wf_prog G⟧ ⟹ is_type G T"
by (auto dest!: wt_is_type)
lemma ty_exprs_is_type:
"⟦(G, L)⊢es::�doteq Ts; wf_prog G⟧ ⟹ Ball (set Ts) (is_type G)"
by (auto dest!: wt_is_type)

lemma static_mheadsD:
 "⟦(cT, m, pns, rT) ∈ mheads G t sig; wf_prog G; (G, L)⊢e::-RefT t;
       m ∨ e = Super⟧ ⟹ ∃C D b. t = ClassT C ∧ cT = ClassT D ∧
       cmethd G C sig = Some (D, (m, pns, rT), b)"
apply (frule (1) ty_expr_is_type)
apply (case_tac "m")
apply  (force dest!: mheadsD imethds_wf_mhead cmethd_wf_mdecl)
apply (clarsimp, erule wt_elim_cases)
apply (auto simp add: cmheads_def)
done

lemma wt_MethdI:
"⟦cmethd G C sig = Some (md, (m, pns, rT), lvars, blk, res); wf_prog G;

  class G C = Some y⟧ ⟹
 ∃T. (G, table_of lvars(pns[↦]snd sig) (+)
```

```
        (if m then empty else empty(()↦Class md)))⊢Methd C sig::-T ∧
G⊢T⪯rT"
apply (frule (2) cmethd_wf_mdecl, clarify)
apply (force dest!: wf_mdecl_bodyD intro!: wt.Methd)
done

end
```

```
theory State = TypeRel:
```

33 objects

```
datatype  obj_tag =
          CInst tname
        | Arr  ty int


types   vn   = "fspec + int"
        obj  = "obj_tag × (vn, val) table"

constdefs

  the_Arr :: "obj option ⇒ ty × int × (vn, val) table"
 "the_Arr obj ≡ ε(T,k,t). obj = Some (Arr T k,t)"

lemma the_Arr_Arr [simp]: "the_Arr (Some (Arr T k,cs)) = (T,k,cs)"
apply (auto simp: the_Arr_def)
done

constdefs

  upd_obj       :: "vn ⇒ val ⇒ obj ⇒ obj"
 "upd_obj n v ≡ λ(oi,vs). (oi,vs(n↦v))"

lemma upd_obj_def2 [simp]: "upd_obj n v (oi, vs) = (oi, vs(n↦v))"
apply (auto simp: upd_obj_def)
done

constdefs
  obj_ty        :: "obj ⇒ ty"
 "obj_ty obj    ≡ case fst obj of CInst C ⇒ Class C | Arr T k ⇒ T.[]"

lemma obj_ty_eq [intro!]: "obj_ty (oi,x) = obj_ty (oi,y)"
apply (simp add: obj_ty_def)
done

lemma obj_ty_cong [simp]: "obj_ty (oi, vs(n↦v)) = obj_ty (oi, vs)"
```

apply *(rule obj_ty_eq)*
done

lemma *obj_ty_CInst [simp]: "obj_ty (CInst C,x) = Class C"*
apply *(simp add: obj_ty_def)*
done

lemma *obj_ty_Arr [simp]: "obj_ty (Arr T i,x) = T.[]"*
apply *(simp add: obj_ty_def)*
done

lemma *obj_ty_widenD: "G⊢obj_ty (oi, vs)≼RefT t ⟹ (∃C. oi = CInst C) ∨ (∃T k. oi = Arr T k)"*
apply *(unfold obj_ty_def)*
apply *(auto split add: obj_tag.split_asm)*
done

constdefs

obj_class :: "obj ⇒ tname"
"obj_class obj ≡ case fst obj of CInst C ⇒ C | Arr T k ⇒ Object"

lemma *obj_class_CInst [simp]: "obj_class (CInst C,fs) = C"*
apply *(auto simp: obj_class_def)*
done

lemma *obj_class_Arr [simp]: "obj_class (Arr T k,fs) = Object"*
apply *(auto simp: obj_class_def)*
done

34 object references

types *oref = "loc + tname"*
syntax
Heap :: "loc ⇒ oref"
Stat :: "tname ⇒ oref"

translations
"Heap" => "Inl"
"Stat" => "Inr"

constdefs
fields_table :: "prog ⇒ tname ⇒ (fspec ⇒ field ⇒ bool) ⇒ (fspec, ty) table"
"fields_table G C P ≡ option_map snd ∘ table_of (filter (split P) (fields G C))"

lemma *fields_table_SomeI: "⟦table_of (fields G C) n = Some (m,T); P n (m,T)⟧ ⟹*

```
  fields_table G C P n = Some T"
apply (unfold fields_table_def)
apply clarsimp
apply (rule exI)
apply (erule map_of_filter_in)
apply assumption
done


lemma fields_table_SomeD': "fields_table G C P fn = Some T ⟹
  ∃m. (fn,(m,T))∈set(fields G C)"
apply (unfold fields_table_def)
apply clarsimp
apply (drule map_of_SomeD)
apply auto
done

lemma fields_table_SomeD:
"⟦fields_table G C P fn = Some T; unique (fields G C)⟧ ⟹
  ∃m. table_of (fields G C) fn = Some (m,T)"
apply (unfold fields_table_def)
apply clarsimp
apply (rule exI)
apply (erule table_of_filter_unique_SomeD)
apply assumption
done

constdefs
  in_bounds :: "int ⇒ int ⇒ bool"            ("(_/ in'_bounds _)" [50,
51] 50)
 "i in_bounds k ≡ 0 ≤ i ∧ i < k"

  arr_comps :: "'a ⇒ int ⇒ int ⇒ 'a option"
 "arr_comps T k ≡ λi. if i in_bounds k then Some T else None"

  var_tys       :: "prog ⇒ obj_tag ⇒ oref ⇒ (vn, ty) table"
 "var_tys G oi r ≡ case r of Heap a ⇒ (case oi of
           CInst C ⇒ fields_table G C (λn (m,fT). ¬static m) (+) empty
         | Arr T k ⇒ empty (+) arr_comps T k)
                         | Stat C ⇒ fields_table G C
                              (λ(fn,fd) (m,fT). fd = C ∧ static m)
(+) empty"

lemma var_tys_Some_eq:
 "var_tys G oi r n = Some T = (case r of Inl a ⇒ (case oi of
    CInst C ⇒ (∃nt. n = Inl nt ∧ fields_table G C (λn (m,fT). ¬static
m) nt = Some T)
  | Arr t k ⇒ (∃ i. n = Inr i  ∧ i in_bounds k ∧ t = T))
  | Inr C   ⇒ (∃nt. n = Inl nt ∧ fields_table G C (λ(fn,fd) (m,fT).
```

```
fd = C ∧ static m) nt = Some T))"
```
apply *(unfold var_tys_def arr_comps_def)*
apply *(force split add: sum.split_asm sum.split obj_tag.split)*
done

35 stores

types *globs*
= "(oref , obj) table"
heap
= "(loc , obj) table"
locals
= "(lname, val) table"

datatype *st =*
st globs locals

35.1 access

constdefs

```
 globs  :: "st ⇒ globs"
"globs  ≡ st_case (λg l. g)"

 locals :: "st ⇒ locals"
"locals ≡ st_case (λg l. l)"

 heap   :: "st ⇒ heap"
"heap s ≡ globs s ∘ Heap"
```

lemma *globs_def2 [simp]: " globs (st g l) = g"*
by *(simp add: globs_def)*

lemma *locals_def2 [simp]: "locals (st g l) = l"*
by *(simp add: locals_def)*

lemma *heap_def2 [simp]: "heap s a=globs s (Heap a)"*
by *(simp add: heap_def)*

syntax

```
  val_this     :: "st ⇒ val"
  lookup_obj   :: "st ⇒ val ⇒ obj"
```

translations

```
"val_this s"       == "the (locals s This)"
"lookup_obj s a'"  == "the (heap s (the_Addr a'))"
```

35.2 memory allocation

constdefs

```
  new_Addr     :: "heap ⇒ loc option"
 "new_Addr h   ≡ if (∀a. h a ≠ None) then None else Some (εa. h a =
None)"
```

lemma *new_AddrD: "new_Addr h = Some a ⟹ h a = None"*
apply *(unfold new_Addr_def)*
apply *auto*
apply *(drule split_paired_All [THEN iffD2])*
apply *(fast intro: someI2)*
done

lemma *new_AddrD2: "new_Addr h = Some a ⟹ ∀b. h b ≠ None ⟶ b ≠
a"*
apply *(drule new_AddrD)*
apply *auto*
done

lemma *new_Addr_SomeI: "h a = None ⟹ ∃b. new_Addr h = Some b ∧ h b
= None"*
apply *(unfold new_Addr_def)*
apply *(frule not_Some_eq [THEN iffD2])*
apply *auto*
apply *(drule not_Some_eq [THEN iffD2])*
apply *auto*
apply *(fast intro!: someI2)*
done

35.3 initialization

syntax

```
  init_vals     :: "('a, ty) table ⇒ ('a, val) table"
```

translations

```
 "init_vals vs"    == "option_map default_val ∘ vs"
```

lemma *init_arr_comps_base [simp]: "init_vals (arr_comps T #0) = empty"*
apply *(unfold arr_comps_def in_bounds_def)*
apply *(rule ext)*
apply *auto*
done
lemma *init_arr_comps_step [simp]:*
"0 < j ⟹ init_vals (arr_comps T j   ) =
*   init_vals (arr_comps T (j-#1))(j-#1↦default_val T)"*
apply *(unfold arr_comps_def in_bounds_def)*
apply *(rule ext)*
apply *auto*

done

35.4 update

constdefs

```
  gupd       :: "oref  ⇒ obj ⇒ st ⇒ st"        ("gupd'(_↦_')"[10,10]1000)
 "gupd r obj  ≡ st_case (λg l. st (g(r↦obj)) l)"

  lupd       :: "lname ⇒ val ⇒ st ⇒ st"        ("lupd'(_↦_')"[10,10]1000)
 "lupd vn v   ≡ st_case (λg l. st g (l(vn↦v)))"

  upd_gobj   :: "oref ⇒ vn ⇒ val ⇒ st ⇒ st"
 "upd_gobj r n v ≡ st_case (λg l. st (chg_map (upd_obj n v) r g) l)"

  set_locals  :: "locals ⇒ st ⇒ st"
 "set_locals l ≡ st_case (λg l'. st g l)"

  init_obj    :: "prog ⇒ obj_tag ⇒ oref ⇒ st ⇒ st"
 "init_obj G oi r ≡ gupd(r↦(oi, init_vals (var_tys G oi r)))"
```

syntax

```
  init_class_obj :: "prog ⇒ tname ⇒ st ⇒ st"
```

translations

```
 "init_class_obj G C" == "init_obj G arbitrary (Inr C)"
```

lemma *gupd_def2 [simp]: "gupd(r↦obj) (st g l) = st (g(r↦obj)) l"*
apply *(unfold gupd_def)*
apply *(simp (no_asm))*
done

lemma *lupd_def2 [simp]: "lupd(vn↦v) (st g l) = st g (l(vn↦v))"*
apply *(unfold lupd_def)*
apply *(simp (no_asm))*
done

lemma *globs_gupd [simp]: "globs (gupd(r↦obj) s) = globs s(r↦obj)"*
apply *(induct "s")*
by *(simp add: gupd_def)*

lemma *globs_lupd [simp]: "globs (lupd(vn↦v) s) = globs s"*
apply *(induct "s")*
by *(simp add: lupd_def)*

lemma *locals_gupd [simp]: "locals (gupd(r↦obj) s) = locals s"*
apply *(induct "s")*
by *(simp add: gupd_def)*

lemma *locals_lupd [simp]: "locals (lupd(vn↦v) s) = locals s(vn↦v*

```
)"
apply (induct "s")
by (simp add: lupd_def)

lemma globs_upd_gobj_new [rule_format (no_asm), simp]:
  "globs s r = None ⟶ globs (upd_gobj r n v s) = globs s"
apply (unfold upd_gobj_def)
apply (induct "s")
apply auto
done

lemma globs_upd_gobj_upd [rule_format (no_asm), simp]:
"globs s r=Some obj ⟶ globs (upd_gobj r n v s) = globs s(r↦upd_obj n
v obj)"
apply (unfold upd_gobj_def)
apply (induct "s")
apply auto
done

lemma locals_upd_gobj [simp]: "locals (upd_gobj r n v s) = locals s"
apply (induct "s")
by (simp add: upd_gobj_def)


lemma globs_init_obj [simp]: "globs (init_obj G oi r s) t =
  (if t=r then Some (oi, init_vals (var_tys G oi r)) else globs s t)"
apply (unfold init_obj_def)
apply (simp (no_asm))
done

lemma locals_init_obj [simp]: "locals (init_obj G oi r s) = locals s"
by (simp add: init_obj_def)

lemma surjective_st [simp]: "st (globs s) (locals s) = s"
apply (induct "s")
by auto

lemma surjective_st_init_obj:
 "st (globs (init_obj G oi r s)) (locals s) = init_obj G oi r s"
apply (subst locals_init_obj [THEN sym])
apply (rule surjective_st)
done

lemma heap_heap_upd [simp]: "heap (st (g(Inl a↦obj)) l) = heap (st g
l)(a↦obj)"
apply (rule ext)
apply (simp (no_asm))
done
lemma heap_stat_upd [simp]: "heap (st (g(Inr C↦obj)) l) = heap (st g
```

```
l)"
apply (rule ext)
apply (simp (no_asm))
done
lemma heap_local_upd [simp]: "heap (st g (l(vn↦v))) = heap (st g l)"
apply (rule ext)
apply (simp (no_asm))
done

lemma heap_gupd_Heap [simp]: "heap (gupd(Heap a↦obj) s) = heap s(a↦obj)"
apply (rule ext)
apply (simp (no_asm))
done
lemma heap_gupd_Stat [simp]: "heap (gupd(Stat C↦obj) s) = heap s"
apply (rule ext)
apply (simp (no_asm))
done
lemma heap_lupd [simp]: "heap (lupd(vn↦v) s) = heap s"
apply (rule ext)
apply (simp (no_asm))
done



lemma heap_upd_gobj_Stat [simp]: "heap (upd_gobj (Stat C) n v s) = heap
s"
apply (rule ext)
apply (simp (no_asm))
apply (case_tac "globs s (Stat C)")
apply  auto
done

lemma set_locals_def2 [simp]: "set_locals l (st g l') = st g l"
apply (unfold set_locals_def)
apply (simp (no_asm))
done

lemma set_locals_id [simp]: "set_locals (locals s) s = s"
apply (unfold set_locals_def)
apply (induct_tac "s")
apply (simp (no_asm))
done

lemma set_set_locals [simp]: "set_locals l (set_locals l' s) = set_locals
l s"
apply (unfold set_locals_def)
apply (induct_tac "s")
apply (simp (no_asm))
done
```

lemma *locals_set_locals [simp]: "locals (set_locals l s) = l"*
apply *(unfold set_locals_def)*
apply *(induct_tac "s")*
apply *(simp (no_asm))*
done

lemma *globs_set_locals [simp]: "globs (set_locals l s) = globs s"*
apply *(unfold set_locals_def)*
apply *(induct_tac "s")*
apply *(simp (no_asm))*
done

lemma *heap_set_locals [simp]: "heap (set_locals l s) = heap s"*
apply *(unfold heap_def)*
apply *(induct_tac "s")*
apply *(simp (no_asm))*
done

36 exceptions

datatype *xcpt*
= XcptLoc loc
| StdXcpt xname

consts

the_XcptLoc :: "xcpt ⇒ loc"
the_StdXcpt :: "xcpt ⇒ xname"

primrec *"the_XcptLoc (XcptLoc a) = a"*
primrec *"the_StdXcpt (StdXcpt x) = x"*

types
xopt = "xcpt option"

constdefs
xcpt_if :: "bool ⇒ xopt ⇒ xopt ⇒ xopt"
"xcpt_if c x' x ≡ if c ∧ (x = None) then x' else x"

lemma *xcpt_if_True_None [simp]: "xcpt_if True x None = x"*
by *(simp add: xcpt_if_def)*

lemma *xcpt_if_True_not_None [simp]: "x ≠ None ⟹ xcpt_if True x y ≠ None"*
by *(simp add: xcpt_if_def)*

lemma *xcpt_if_False [simp]: "xcpt_if False x y = y"*

by *(simp add: xcpt_if_def)*

lemma *xcpt_if_Some [simp]: "xcpt_if c x (Some y) = Some y"*
by *(simp add: xcpt_if_def)*

lemma *xcpt_if_not_None [simp]: "y ≠ None ⟹ xcpt_if c x y = y"*
apply *(simp add: xcpt_if_def)*
by *auto*

lemma *split_xcpt_if:*
"P (xcpt_if c x' x) = ((c ∧ x = None ⟶ P x') ∧ (¬ (c ∧ x = None) ⟶ P x))"
apply *(unfold xcpt_if_def)*
apply *(split split_if)*
apply *auto*
done

syntax

```
raise_if :: "bool ⇒ xname ⇒ xopt ⇒ xopt"
np       :: "val          ⇒ xopt ⇒ xopt"
check_neg:: "val          ⇒ xopt ⇒ xopt"
```

translations

```
"raise_if c xn" == "xcpt_if c (Some (StdXcpt xn))"
"np v"          == "raise_if (v = Null)      NullPointer"
"check_neg i'"  == "raise_if (the_Intg i'<0) NegArrSize"
```

lemma *raise_if_None [simp]: "(raise_if c x y = None) = (¬c ∧ y = None)"*
apply *(simp add: xcpt_if_def)*
by *auto*
declare *raise_if_None [THEN iffD1, dest!]*

lemma *if_raise_if_None [simp]:*
"((if b then y else raise_if c x y) = None) = ((c ⟶ b) ∧ y = None)"
apply *(simp add: xcpt_if_def)*
apply *auto*
done

lemma *raise_if_SomeD [dest!]:*
"raise_if c x y = Some z ⟹ c ∧ z=StdXcpt x ∧ y=None ∨ (y=Some z)"
apply *(case_tac y)*
apply *(case_tac c)*
apply *(simp add: xcpt_if_def)*
apply *(simp add: xcpt_if_def)*
apply *auto*
done

37 full program state

types
```
  state = "xopt × st"
```

syntax

```
  Norm     :: "st ⇒ state"
```

translations

```
  "Norm s"     == "(None,s)"
  "xopt"       <= (type) "State.xcpt option"
  "xopt"       <= (type) "xcpt option"
  "state"      <= (type) "xopt × State.st"
  "state"      <= (type) "xopt × st"
```

lemma *single_stateE: "∀ Z. Z = (s::state) ⟹ False"*
apply *(erule_tac x = "(Some k,y)"* **in** *all_dupE)*
apply *(erule_tac x = "(None ,y)"* **in** *allE)*
apply *clarify*
done

lemma *state_not_single: "All (op = (x::('a option) × 'b)) ⟹ R"*
apply *(drule_tac x = "(if fst x = None then Some ?x else None,?y)"* **in** *spec)*
apply *clarsimp*
done

constdefs

```
  normal     :: "state ⇒ bool"
 "normal ≡ λs. fst s = None"
```

lemma *normal_def2 [simp]: "normal s = (fst s = None)"*
apply *(unfold normal_def)*
apply *(simp (no_asm))*
done

constdefs
```
  heap_free :: "nat ⇒ state ⇒ bool"
 "heap_free n ≡ λs. atleast_free (heap (snd s)) n"
```

lemma *heap_free_def2 [simp]: "heap_free n s = atleast_free (heap (snd s)) n"*
apply *(unfold heap_free_def)*
apply *simp*
done

37.1 update

constdefs

```
  xupd     :: "(xopt ⇒ xopt) ⇒ state ⇒ state"
 "xupd f ≡ prod_fun f id"

  supd     :: "(st ⇒ st) ⇒ state ⇒ state"
 "supd ≡ prod_fun id"
```

lemma *xupd_def2 [simp]: "xupd f (x,s) = (f x,s)"*
by *(simp add: xupd_def)*

lemma *xupd_xcpt_if_False [simp]: "⋀ s. xupd (xcpt_if False xo) s = s"*
by *simp*

lemma *supd_def2 [simp]: "supd f (x,s) = (x,f s)"*
by *(simp add: supd_def)*

lemma *supd_lupd [simp]: "⋀ s. supd (lupd vn v) s = (fst s,lupd vn v (snd s))"*
apply *(simp (no_asm_simp) only: split_tupled_all)*
apply *(simp (no_asm))*
done

lemma *supd_gupd [simp]: "⋀ s. supd (gupd r obj) s = (fst s,gupd r obj (snd s))"*
apply *(simp (no_asm_simp) only: split_tupled_all)*
apply *(simp (no_asm))*
done

lemma *supd_init_obj [simp]:*
"supd (init_obj G oi r) s = (fst s, init_obj G oi r (snd s))"
apply *(unfold init_obj_def)*
apply *(simp (no_asm))*
done

syntax

```
  set_lvars     :: "locals ⇒ state ⇒ state"
  restore_lvars :: "state  ⇒ state ⇒ state"
```

translations

```
 "set_lvars l" == "supd (set_locals l)"
 "restore_lvars s' s" == "set_lvars (locals (snd s')) s"
```

lemma *set_set_lvars [simp]: "⋀ s. set_lvars l (set_lvars l' s) = set_lvars l s"*

apply *(simp (no_asm_simp) only: split_tupled_all)*
apply *(simp (no_asm))*
done

lemma *set_lvars_id [simp]: "⋀ s. set_lvars (locals (snd s)) s = s"*
apply *(simp (no_asm_simp) only: split_tupled_all)*
apply *(simp (no_asm))*
done

38 initialisation test

constdefs

inited :: "tname ⇒ globs ⇒ bool"
"inited C g ≡ g (Stat C) ≠ None"

initd :: "tname ⇒ state ⇒ bool"
"initd C ≡ inited C ∘ globs ∘ snd"

lemma *not_inited_empty [simp]: "¬inited C empty"*
apply *(unfold inited_def)*
apply *(simp (no_asm))*
done

lemma *inited_gupdate [simp]: "inited C (g(r↦obj)) = (inited C g ∨ r = Stat C)"*
apply *(unfold inited_def)*
apply *(auto split add: st.split)*
done

lemma *inited_init_class_obj [intro!]: "inited C (globs (init_class_obj G C s))"*
apply *(unfold inited_def)*
apply *(simp (no_asm))*
done

lemma *not_initedD: "¬ inited C g ⟹ g (Stat C) = None"*
apply *(unfold inited_def)*
apply *(erule notnotD)*
done

lemma *initedD: "inited C g ⟹ ∃oi fs. g (Stat C) = Some (oi, fs)"*
apply *(unfold inited_def)*
apply *auto*
done

lemma *initd_def2 [simp]: "initd C s = inited C (globs (snd s))"*
apply *(unfold initd_def)*
apply *(simp (no_asm))*

done

end

theory *Eval = State:*

types *vvar = "val × (val ⇒ state ⇒ state)"*
vals = "(val, vvar, val list) sum3"
translations
"vvar" <= (type) "val × (val ⇒ state ⇒ state)"
"vals" <= (type)"(val, vvar, val list) sum3"

syntax *(xsymbols)*
dummy_res :: "vals" ("•")
translations
"•" == "In1 Unit"

constdefs
arbitrary3 :: "('al + 'ar, 'b, 'c) sum3 ⇒ vals"
"arbitrary3 ≡ sum3_case (In1 ∘ sum_case (λx. arbitrary) (λx. Unit))
(λx. In2 arbitrary) (λx. In3 arbitrary)"

lemma *[simp]: "arbitrary3 (In1l x) = In1 arbitrary"*
by *(simp add: arbitrary3_def)*

lemma *[simp]: "arbitrary3 (In1r x) = •"*
by *(simp add: arbitrary3_def)*

lemma *[simp]: "arbitrary3 (In2 x) = In2 arbitrary"*
by *(simp add: arbitrary3_def)*

lemma *[simp]: "arbitrary3 (In3 x) = In3 arbitrary"*
by *(simp add: arbitrary3_def)*

39 exception throwing and catching

constdefs
throw :: "val ⇒ xopt ⇒ xopt"
"throw a' x ≡ xcpt_if True (Some (XcptLoc (the_Addr a'))) (np a' x)"

lemma *throw_def2:*
"throw a' x = xcpt_if True (Some (XcptLoc (the_Addr a'))) (np a' x)"
apply *(unfold throw_def)*
apply *(simp (no_asm))*
done

constdefs

```
  fits    :: "prog ⇒ st ⇒ val ⇒ ty ⇒ bool" ("_,_⊢_ fits _"[61,61,61,61]60)
 "G,s⊢a' fits T  ≡ (∃rt. T=RefT rt) ⟶ a'=Null ∨ G⊢obj_ty(lookup_obj
s a')⪯T"

lemma fits_Null [simp]: "G,s⊢Null fits T"
by (simp add: fits_def)


lemma fits_Addr_RefT [simp]:
  "G,s⊢Addr a fits RefT t = G⊢obj_ty (the (heap s a))⪯RefT t"
by (simp add: fits_def)

lemma fitsD: "⋀X. G,s⊢a' fits T ⟹ (∃pt. T = PrimT pt) ∨
  (∃t. T = RefT t) ∧ a' = Null ∨
  (∃t. T = RefT t) ∧ a' ≠ Null ∧  G⊢obj_ty (lookup_obj s a')⪯T"
apply (unfold fits_def)
apply (case_tac "∃pt. T = PrimT pt")
apply  simp_all
apply (case_tac "T")
defer
apply (case_tac "a' = Null")
apply  simp_all
done

constdefs
  catch ::"prog ⇒ state ⇒ tname ⇒ bool"      ("_,_⊢catch _"[61,61,61]60)
 "G,s⊢catch C≡∃xc. fst s=Some xc ∧ G,snd s⊢Addr (the_XcptLoc xc) fits
Class C"

lemma catch_Norm [simp]: "¬G,Norm s⊢catch tn"
apply (unfold catch_def)
apply (simp (no_asm))
done

lemma catch_XcptLoc [simp]: "G,(Some (XcptLoc a), s)⊢catch C = G,s⊢Addr
a fits Class C"
apply (unfold catch_def)
apply (simp (no_asm))
done

constdefs
  new_xcpt_var :: "ename ⇒ state ⇒ state"
 "new_xcpt_var vn ≡ λ(x,s). Norm(lupd(EName vn↦Addr (the_XcptLoc (the
x))) s)"

lemma new_xcpt_var_def2 [simp]:
  "new_xcpt_var vn (x,s) = Norm (lupd(EName vn↦Addr (the_XcptLoc (the
x))) s)"
apply (unfold new_xcpt_var_def)
```

apply *(simp (no_asm))*
done

40 misc

constdefs

```
  assign     :: "('a ⇒ state ⇒ state) ⇒ 'a ⇒ state ⇒ state"
 "assign f v ≡ λ(x,s). let (x',s') = (if x = None then f v else id) (x,s)
                  in  (x',if x' = None then s' else s)"
```

lemma *assign_Norm_Norm [simp]:*
"f v (Norm s) = Norm s' ⟹ assign f v (Norm s) = Norm s'"
by *(simp add: assign_def Let_def)*

lemma *assign_Norm_Some [simp]:*
"⟦fst (f v (Norm s)) = Some y⟧ ⟹ assign f v (Norm s) = (Some y,s)"
by *(simp add: assign_def Let_def split_beta)*

lemma *assign_Some [simp]: "assign f v (Some x,s) = (Some x,s)"*
by *(simp add: assign_def Let_def split_beta)*

lemma *assign_supd [simp]:*
"assign (λv. supd (f v)) v (x, s) = (x, if x = None then f v s else s)"
by *auto*

lemma *assign_raise_if [simp]:*
"assign (λv (x,s). ((raise_if (b s v) xcpt) x, f v s)) v (x, s) =
(raise_if (b s v) xcpt x, if x=None ∧ ¬b s v then f v s else s)"
apply *(case_tac "x = None")*
apply *auto*
done

constdefs

```
  init_comp_ty :: "ty ⇒ stmt"
 "init_comp_ty T ≡ if (∃C. T = Class C) then init (the_Class T) else
Skip"
```

lemma *init_comp_ty_PrimT [simp]: "init_comp_ty (PrimT pt) = Skip"*
apply *(unfold init_comp_ty_def)*
apply *(simp (no_asm))*
done

constdefs

```
  target  :: "inv_mode ⇒ st ⇒ val ⇒ ref_ty ⇒ tname"
 "target m s a' t ≡ if m = IntVir
         then obj_class (lookup_obj s a') else the_Class (RefT t)"
```

```
lemma target_IntVir [simp]:
"target IntVir s a' t = obj_class (lookup_obj s a')"
apply (unfold target_def)
apply (simp (no_asm))
done

lemma target_notIntVir: "m ≠ IntVir ⟹ target m s a' t = the_Class
(RefT t)"
apply (unfold target_def)
apply (simp (no_asm_simp))
done

lemma target_Static [simp]: "target Static s a' t = the_Class (RefT t)"
apply (unfold target_def)
apply (simp (no_asm))
done

lemma target_SuperM [simp]: "target SuperM s a' t = the_Class (RefT t)"
apply (unfold target_def)
apply (simp (no_asm))
done


constdefs
  init_lvars :: "prog ⇒ tname ⇒ sig ⇒ inv_mode ⇒ val ⇒ val list
⇒
                   state ⇒ state"
 "init_lvars G C sig mode a' pvs ≡ λ(x,s). let
              (_,(_,pns,_),lvars,_) = the (cmethd G C sig);
              l = init_vals(table_of lvars)(pns[↦]pvs) (+)
                  (if mode=Static then empty else empty(()↦a'))
              in set_lvars l (if mode = Static then x else np a' x,s)"

lemma init_lvars_def2: "init_lvars G C sig mode a' pvs (x,s) =
  set_lvars (init_vals(table_of (fst (snd (snd( the (cmethd G C sig))))))(fst
(snd (fst (snd (the (cmethd G C sig)))))[↦]pvs) (+)
  (if mode=Static then empty else empty(()↦a')))
  (if mode=Static then x     else np a' x, s)"
apply (unfold init_lvars_def)
apply (simp (no_asm) add: Let_def split_beta)
done

constdefs
  body :: "prog ⇒ tname ⇒ sig ⇒ expr"
 "body G C sig ≡ let (D, _, _, c, e) = the (cmethd G C sig) in Body D
c e"

lemma body_def2:
```

```
"body G C sig = (λ (D, x1, x2, c, ee). Body D c ee) (the (cmethd G C sig))"
```
apply *(unfold body_def Let_def)*
apply *auto*
done

41 variables

constdefs

```
  lvar :: "lname ⇒ st ⇒ vvar"
 "lvar vn s ≡ (the (locals s vn), λv. supd (lupd(vn↦v)))"

  fvar :: "tname ⇒ bool ⇒ ename ⇒ val ⇒ state ⇒ vvar × state"
 "fvar C stat fn a' s ≡ let (oref,xf) = if stat then (Stat C,id)
                                        else (Heap (the_Addr a'),np a');
                        n = Inl (fn,C); f = (λv. supd (upd_gobj oref n
v)) in
                  ((the (snd (the (globs (snd s) oref)) n),f),xupd xf
s)"

  avar :: "prog ⇒ val ⇒ val ⇒ state ⇒ vvar × state"
 "avar G i' a' s ≡ let oref = Heap (the_Addr a'); i = the_Intg i'; n
= Inr i;
                       (T,k,cs) = the_Arr (globs (snd s) oref); f = (λv
(x,s).
             (raise_if (¬G,s⊢v fits T) ArrStore x, upd_gobj oref n v
s)) in
       ((the (cs n),f), xupd (raise_if (¬i in_bounds k) IndOutBound ∘
np a') s)"
```

lemma *fvar_def2: "fvar C stat fn a' s =*
```
  ((the (snd (the (globs(snd s) (if stat then Stat C else (Heap(the_Addr
a')))))
  (Inl (fn,C))),(λv. supd (upd_gobj (if stat then Stat C else
  (Heap (the_Addr a'))) (Inl (fn,C)) v))),xupd (if stat then id else np
a') s)"
```
apply *(unfold fvar_def)*
apply *(simp (no_asm) add: Let_def split_beta)*
done

lemma *avar_def2: "avar G i' a' s =*
```
 ((the(snd(snd(the_Arr(globs (snd s)(Heap(the_Addr a')))))) (Inr(the_Intg
i'))),
 (λv (x,s').
   (raise_if (¬G,s'⊢v fits(fst(the_Arr(globs (snd s)(Heap(the_Addr a'))))))
             ArrStore x,
  upd_gobj (Heap (the_Addr a')) (Inr (the_Intg i')) v s')
  )),
   xupd (raise_if (¬ the_Intg i' in_bounds
```

```
   fst(snd((the_Arr(globs (snd s) (Heap (the_Addr a')))))) IndOutBound

   ∘ np a') s)"
apply (unfold avar_def)
apply (simp (no_asm) add: Let_def split_beta)
done
```

42 evaluation judgments

consts

```
  eval   :: "prog ⇒ (state × term    × vals × state) set"
   halloc:: "prog ⇒ (state × obj_tag × loc  × state) set"
  sxalloc:: "prog ⇒ (state                 × state) set"
```

syntax

```
eval ::"[prog,state,term,vals*state]=>bool"("_|-_ -_->- _"  [61,61,80,
61]60)
exec ::"[prog,state,stmt      ,state]=>bool"("_|-_ -_-> _"   [61,61,65,
61]60)
evar ::"[prog,state,var  ,vvar,state]=>bool"("_|-_ -_=>_-> _"[61,61,90,61,61]60)
eval_::"[prog,state,expr ,val, state]=>bool"("_|-_ -_->_-> _"[61,61,80,61,61]60)
evals::"[prog,state,expr list ,
                    val  list ,state]=>bool"("_|-_ -_#>_-> _"[61,61,61,61,61]60)
hallo::"[prog,state,obj_tag,
                     loc,state]=>bool"("_|-_ -halloc _>_-> _"[61,61,61,61,61]60)
sallo::"[prog,state         ,state]=>bool"("_|-_ -sxalloc-> _"[61,61,
61]60)
```

syntax *(xsymbols)*

```
eval ::"[prog,state,term,vals×state]⇒bool" ("_⊢_ −_≻→ _"  [61,61,80,
61]60)
exec ::"[prog,state,stmt      ,state]⇒bool"("_⊢_ −_→ _"   [61,61,65,
61]60)
evar ::"[prog,state,var  ,vvar,state]⇒bool"("_⊢_ −_=≻_→ _"[61,61,90,61,61]60)
eval_::"[prog,state,expr ,val ,state]⇒bool"("_⊢_ −_-≻_→ _"[61,61,80,61,61]60)
evals::"[prog,state,expr list ,
                    val  list ,state]⇒bool"("_⊢_ −_≐≻_→ _"[61,61,61,61,61]60)
hallo::"[prog,state,obj_tag,
                     loc,state]⇒bool"("_⊢_ −halloc _≻_→ _"[61,61,61,61,61]60)
sallo::"[prog,state,         state]⇒bool"("_⊢_ −sxalloc→ _"[61,61,
61]60)
```

translations

```
  "G⊢s −t   ≻→  w___s' " == "(s,t,w___s') ∈ eval G"
  "G⊢s −t   ≻→ (w,  s')" <= "(s,t,w,  s') ∈ eval G"
  "G⊢s −t   ≻→ (w,x,s')" <= "(s,t,w,x,s') ∈ eval G"
  "G⊢s −c    →  (x,s')" <= "G⊢s −In1r c≻→ (•     ,x,s')"
  "G⊢s −c    →     s' " == "G⊢s −In1r c≻→ (•     ,  s')"
```

```
"G⊢s −e-≻v → (x,s')" <= "G⊢s −In1l e≻→ (In1 v ,x,s')"
"G⊢s −e-≻v →    s' " == "G⊢s −In1l e≻→ (In1 v ,  s')"
"G⊢s −e=≻vf→ (x,s')" <= "G⊢s −In2  e≻→ (In2 vf,x,s')"
"G⊢s −e=≻vf→    s' " == "G⊢s −In2  e≻→ (In2 vf,  s')"
"G⊢s −e≐≻v → (x,s')" <= "G⊢s −In3  e≻→ (In3 v ,x,s')"
"G⊢s −e≐≻v →    s' " == "G⊢s −In3  e≻→ (In3 v ,  s')"
"G⊢s −halloc oi≻a→ (x,s')" <= "(s,oi,a,x,s') ∈ halloc G"
"G⊢s −halloc oi≻a→    s' " == "(s,oi,a,  s') ∈ halloc G"
"G⊢s −sxalloc→     (x,s')" <= "(s     ,x,s') ∈ sxalloc G"
"G⊢s −sxalloc→        s' " == "(s     ,  s') ∈ sxalloc G"
```

inductive *"halloc G"* **intros**

```
Xcpt: "G⊢(Some x,s) −halloc oi≻arbitrary→ (Some x,s)"

New:  "⟦new_Addr (heap s) = Some a;
        (x,oi') = (if atleast_free (heap s) 2 then (None,oi)
                   else (Some (XcptLoc a),CInst (SXcpt OutOfMemory)))⟧
⟹
        G⊢Norm s −halloc oi≻a→ (x,init_obj G oi' (Heap a) s)"
```

inductive *"sxalloc G"* **intros**

```
Norm:  "G⊢ Norm              s   −sxalloc→  Norm             s"

XcptL: "G⊢(Some (XcptLoc a ),s)  −sxalloc→ (Some (XcptLoc a),s)"

SXcpt: "⟦G⊢Norm s0 −halloc (CInst (SXcpt xn))≻a→ (x,s1)⟧ ⟹
        G⊢(Some (StdXcpt xn),s0) −sxalloc→ (Some (XcptLoc a),s1)"
```

inductive *"eval G"* **intros**

```
Xcpt: "G⊢(Some xc,s) −t≻→ (arbitrary3 t,(Some xc,s))"

Skip:                              "G⊢Norm s −Skip→ Norm s"

Expr: "⟦G⊢Norm s0 −e-≻v→ s1⟧ ⟹
                                   G⊢Norm s0 −Expr e→ s1"
```

```
  Comp: "⟦G⊢Norm s0 −c1 → s1;
          G⊢     s1 −c2 → s2⟧ ⟹
                                G⊢Norm s0 −c1;; c2→ s2"


  If:   "⟦G⊢Norm s0 −e-≻b→ s1;
          G⊢     s1−(if the_Bool b then c1 else c2)→ s2⟧ ⟹
                      G⊢Norm s0 −If(e) c1 Else c2 → s2"


  Loop: "⟦G⊢Norm s0 −e-≻b→ s1;
          if the_Bool b then (G⊢s1 −c→ s2 ∧ G⊢s2 −While(e) c→ s3)
                       else s3 = s1⟧ ⟹
                             G⊢Norm s0 −While(e) c→ s3"


  Throw: "⟦G⊢Norm s0 −e-≻a'→ s1⟧ ⟹
                                G⊢Norm s0 −Throw e→ xupd (throw a')
s1"


  Try:  "⟦G⊢Norm s0 −c1→ s1; G⊢s1 −sxalloc→ s2;
          if G,s2⊢catch C then G⊢new_xcpt_var vn s2 −c2→ s3 else s3
= s2⟧ ⟹
                  G⊢Norm s0 −Try c1 Catch(C vn) c2→ s3"


  Fin:  "⟦G⊢Norm s0 −c1→ (x1,s1);
          G⊢Norm s1 −c2→ s2⟧ ⟹
                   G⊢Norm s0 −c1 Finally c2→ xupd (xcpt_if (x1≠None)
x1) s2"


  Init: "⟦the (class G C) = (sc,si,fs,ms,ini);
          if inited C (globs s0) then s3 = Norm s0
          else (G⊢Norm (init_class_obj G C s0)
                  −(if C = Object then Skip else init sc)→ s1 ∧
                G⊢set_lvars empty s1 −ini→ s2 ∧ s3 = restore_lvars
s1 s2)⟧ ⟹
                  G⊢Norm s0 −init C→ s3"




  NewC: "⟦G⊢Norm s0 −init C→ s1;
          G⊢     s1 −halloc (CInst C)≻a→ s2⟧ ⟹
```

```
                                    G⊢Norm s0 −NewC C-≻Addr a→ s2"


  NewA: "⟦G⊢Norm s0 −init_comp_ty T→ s1; G⊢s1 −e-≻i'→ s2;
          G⊢xupd (check_neg i') s2 −halloc (Arr T (the_Intg i'))≻a→
s3⟧ ⟹
                                  G⊢Norm s0 −New T[e]-≻Addr a→ s3"


  Cast: "⟦G⊢Norm s0 −e-≻v→ s1;
          s2 = xupd (raise_if (¬G,snd s1⊢v fits T) ClassCast) s1⟧ ⟹
                                  G⊢Norm s0 −Cast T e-≻v→ s2"


  Inst: "⟦G⊢Norm s0 −e-≻v→ s1;
          b = (v≠Null ∧ G,snd s1⊢v fits RefT T)⟧ ⟹
                                G⊢Norm s0 −e InstOf T-≻Bool b→ s1"


  Lit:  "G⊢Norm s −Lit v-≻v→ Norm s"



  Super: "G⊢Norm s −Super-≻val_this s→ Norm s"


  Acc:  "⟦G⊢Norm s0 −va=≻(v,f)→ s1⟧ ⟹
                                  G⊢Norm s0 −Acc va-≻v→ s1"


  Ass:  "⟦G⊢Norm s0 −va=≻(w,f)→ s1;
          G⊢      s1 −e-≻v  → s2⟧ ⟹
                                   G⊢Norm s0 −va:=e-≻v→ assign f v
s2"


  Cond: "⟦G⊢Norm s0 −e0-≻b→ s1;
          G⊢      s1 −(if the_Bool b then e1 else e2)-≻v→ s2⟧ ⟹
                           G⊢Norm s0 −e0 ? e1 : e2-≻v→ s2"


  Call: "⟦G⊢Norm s0 −e-≻a'→ s1; G⊢s1 −args≐≻vs→ s2;
          C = target mode (snd s2) a' cT;
          G⊢init_lvars G C (mn,pTs) mode a' vs s2 −Methd C (mn,pTs)-≻v→
s3⟧⟹
          G⊢Norm s0 −{t,cT,mode}e..mn({pTs}args)-≻v→ (restore_lvars
s2 s3)"
```

```
Methd:        "⟦G⊢Norm s0 −body G C sig-≻v→ s1⟧ ⟹
                              G⊢Norm s0 −Methd C sig-≻v→ s1"

Body: "⟦G⊢Norm s0 −init D→ s1; G⊢s1 −c→ s2; G⊢s2 −e-≻v→ s3⟧ ⟹
                              G⊢Norm s0 −Body D c e-≻v→ s3"

LVar: "G⊢Norm s −LVar vn=≻lvar vn s→ Norm s"

FVar: "⟦G⊢Norm s0 −init C→ s1; G⊢s1 −e-≻a→ s2;
        (v,s2') = fvar C stat fn a s2⟧ ⟹
        G⊢Norm s0 −{C,stat}e..fn=≻v→ s2'"

AVar: "⟦G⊢ Norm s0 −e1-≻a→ s1; G⊢s1 −e2-≻i→ s2;
        (v,s2') = avar G i a s2⟧ ⟹
                    G⊢Norm s0 −e1.[e2]=≻v→ s2'"

Nil:
                                   "G⊢Norm s0 −[]≐≻[]→ Norm s0"

Cons: "⟦G⊢Norm s0 −e -≻ v → s1;
        G⊢      s1 −es≐≻vs→ s2⟧ ⟹
                                  G⊢Norm s0 −e#es≐≻v#vs→ s2"
```

ML *{*
bind_thm ("eval_induct_", rearrange_prems
[0,1,2,26,27,23,13,14,15,
 16,17,3,4,21,22,19,5,6,
 9,7,11,12,10,18,8,24,25,
 20] (thm "eval.induct"))
}
lemmas *eval_induct = eval_induct_ [split_format* **and and and and and and and and**
   and and *s1 s2* **and and** *s2* **and and and and and and** *s2* **and and**
s2 s3]

declare *split_if [split del] split_if_asm [split del]*

```
        option.split [split del] option.split_asm [split del]
inductive_cases halloc_elim_cases:
  "G⊢(Some xc, s) −halloc oi≻a→ s'"
  "G⊢(Norm     s) −halloc oi≻a→ s'"
inductive_cases sxalloc_elim_cases:
        "G⊢ Norm              s  −sxalloc→ s'"
        "G⊢(Some (XcptLoc a ),s) −sxalloc→ s'"
        "G⊢(Some (StdXcpt xn),s) −sxalloc→ s'"
inductive_cases sxalloc_cases: "G⊢s −sxalloc→ s'"

lemma sxalloc_elim_cases2: "⟦G⊢s −sxalloc→ s';
       ⋀s.   ⟦s' = Norm s⟧ ⟹ P;
       ⋀a s. ⟦s' = (Some (XcptLoc a), s)⟧ ⟹ P
      ⟧ ⟹ P"
apply cut_tac
apply (erule sxalloc_cases)
apply blast+
done

declare not_None_eq [simp del] IntDef.Zero_def [simp del]
declare split_paired_All [simp del] split_paired_Ex [simp del]
ML_setup {*
simpset_ref() := simpset() delloop "split_all_tac"
*}
inductive_cases eval_cases: "G⊢s −t≻→ vs'"

inductive_cases eval_elim_cases:
        "G⊢(Some xc, s) −t≻→  vs'"
        "G⊢Norm s −In1r Skip                    ≻→ xs'"
        "G⊢Norm s −In3  ([])                    ≻→ vs'"
        "G⊢Norm s −In3  (e#es)                  ≻→ vs'"
        "G⊢Norm s −In1l (Lit w)                 ≻→ vs'"
        "G⊢Norm s −In2  (LVar vn)               ≻→ vs'"
        "G⊢Norm s −In1l (Cast T e)              ≻→ vs'"
        "G⊢Norm s −In1l (e InstOf T)            ≻→ vs'"
        "G⊢Norm s −In1l (Super)                 ≻→ vs'"
        "G⊢Norm s −In1l (Acc va)                ≻→ vs'"
        "G⊢Norm s −In1r (Expr e)                ≻→ xs'"
        "G⊢Norm s −In1r (c1;; c2)               ≻→ xs'"
        "G⊢Norm s −In1l (Methd C sig)           ≻→ xs'"
        "G⊢Norm s −In1l (Body D c e)            ≻→ xs'"
        "G⊢Norm s −In1l (e0 ? e1 : e2)          ≻→ vs'"
        "G⊢Norm s −In1r (If(e) c1 Else c2)      ≻→ xs'"
        "G⊢Norm s −In1r (While(e) c)            ≻→ xs'"
        "G⊢Norm s −In1r (c1 Finally c2)         ≻→ xs'"
        "G⊢Norm s −In1r (Throw e)               ≻→ xs'"
        "G⊢Norm s −In1l (NewC C)                ≻→ vs'"
        "G⊢Norm s −In1l (New T[e])              ≻→ vs'"
        "G⊢Norm s −In1l (Ass va e)              ≻→ vs'"
```

```
        "G⊢Norm s −In1r (Try c1 Catch(tn vn) c2)  ≻→ xs'"
        "G⊢Norm s −In2  ({C,stat}e..fn)           ≻→ vs'"
        "G⊢Norm s −In2  (e1.[e2])                 ≻→ vs'"
        "G⊢Norm s −In1l ({t,cT,mode}e..mn({pT}p)) ≻→ vs'"
        "G⊢Norm s −In1r (init C)                  ≻→ xs'"
declare not_None_eq [simp]  IntDef.Zero_def [simp]
declare split_paired_All [simp] split_paired_Ex [simp]
ML_setup {*
simpset_ref() := simpset() addloop ("split_all_tac", split_all_tac)
*}
declare split_if     [split] split_if_asm     [split]
        option.split [split] option.split_asm [split]

lemma eval_Inj_elim: "G⊢s −t≻→ (w,s') ⟹ case t of In1 ec ⇒
  (case ec of Inl e ⇒ (∃v. w = In1 v) | Inr c ⇒ w = •)
  | In2 e ⇒ (∃v. w = In2 v) | In3 e ⇒ (∃v. w = In3 v)"
apply (erule eval_cases)
apply auto
apply (induct_tac "t")
apply (induct_tac "a")
apply auto
done

ML_setup {*
fun eval_fun nam inj rhs =
let
  val name = "eval_" ^ nam ^ "_eq"
  val lhs = "G⊢s −" ^ inj ^ " t≻→ (w, s')"
  val () = qed_goal name (the_context()) (lhs ^ " = (" ^ rhs ^ ")")
        (K [Auto_tac, ALLGOALS (ftac (thm "eval_Inj_elim")) THEN Auto_tac])
  fun is_Inj (Const (inj,_) $ _) = true
    | is_Inj _                   = false
  fun pred (_ $ (Const ("Pair",_) $ _ $
      (Const ("Pair", _) $ _ $ (Const ("Pair", _) $ x $ _ ))) $ _ ) =
is_Inj x
in
  make_simproc name lhs pred (thm name)
end

val eval_expr_proc =eval_fun "expr" "In1l" "∃v.  w=In1 v   ∧ G⊢s −t-≻v
→ s'"
val eval_var_proc  =eval_fun "var"  "In2"  "∃vf. w=In2 vf  ∧ G⊢s −t=≻vf→
s'"
val eval_exprs_proc=eval_fun "exprs""In3"  "∃vs. w=In3 vs  ∧ G⊢s −t≐≻vs→
s'"
val eval_stmt_proc =eval_fun "stmt" "In1r" "     w=• ∧ G⊢s −t    → s'";
Addsimprocs [eval_expr_proc,eval_var_proc,eval_exprs_proc,eval_stmt_proc];
bind_thms ("XcptIs", sum3_instantiate (thm "eval.Xcpt"))
*}
```

declare *halloc.Xcpt [intro!] eval.Xcpt [intro!] XcptIs [intro!]*

lemma *eval_no_xcpt_lemma: "⋀s s'. G⊢s −t≻→ (w,s') ⟹ normal s' ⟶ normal s"*
by *(erule eval_cases, auto)*

lemma *eval_no_xcpt:*
"G⊢(x,s) −t≻→ (w,Norm s') = (x = None ∧ G⊢Norm s −t≻→ (w,Norm s'))"
apply *auto*
apply *(frule eval_no_xcpt_lemma, auto)+*
done

ML *{*
local
  fun is_None (Const ("Option.option.None",_)) = true
    | is_None _ = false
  fun pred (t as (_ $ (Const ("Pair",_) $
     (Const ("Pair", _) $ x $ _) $ _) $ _)) = is_None x
in
  val eval_no_xcpt_proc =
  make_simproc "eval_no_xcpt" "G⊢(x,s) −e≻→ (w,Norm s')" pred
         (thm "eval_no_xcpt")
end;
Addsimprocs [eval_no_xcpt_proc]
}

lemma *eval_xcpt_lemma:*
"G⊢s −t≻→ (v,s') ⟹ fst s=Some xc ⟶ s'= s ∧ v = arbitrary3 t"
by *(erule eval_cases, auto)*

lemma *eval_xcpt: "⋀s'. G⊢(Some xc,s) −t≻→ (w,s') =*
(s'=(Some xc,s) ∧ w=arbitrary3 t ∧
G⊢(Some xc,s) −t≻→ (arbitrary3 t,(Some xc,s)))"
apply *auto*
apply *(frule eval_xcpt_lemma, auto)+*
done

ML *{*
local
  fun is_Some (Const ("Pair",_) $ (Const ("Option.option.Some",_) $ _)$ _) =true
    | is_Some _ = false
  fun pred (_ $ (Const ("Pair",_) $
     _ $ (Const ("Pair", _) $ _ $ (Const ("Pair", _) $ _ $
       x))) $ _) = is_Some x*

```
in
  val eval_xcpt_proc =
  make_simproc "eval_xcpt" "G⊢(Some xc,s) -e≻→ (w,s')" pred (thm "eval_xcpt")
end;
Addsimprocs [eval_xcpt_proc]
*}


lemma LitI: "G⊢s -Lit v-≻(if normal s then v else arbitrary)→ s"
apply (case_tac "s", case_tac "a = None")
by (auto intro!: eval.Lit)

lemma SkipI [intro!]: "G⊢s -Skip→ s"
apply (case_tac "s", case_tac "a = None")
by (auto intro!: eval.Skip)

lemma ExprI: "G⊢s -e-≻v→ s' ⟹ G⊢s -Expr e→ s'"
apply (case_tac "s", case_tac "a = None")
by (auto intro!: eval.Expr)

lemma CompI: "⟦G⊢s -c1→ s1; G⊢s1 -c2→ s2⟧ ⟹ G⊢s -c1;; c2→ s2"
apply (case_tac "s", case_tac "a = None")
by (auto intro!: eval.Comp)

lemma CondI:
  "⋀s1. ⟦G⊢s -e-≻b→ s1; G⊢s1 -(if the_Bool b then e1 else e2)-≻v→
s2⟧ ⟹
        G⊢s -e ? e1 : e2-≻(if normal s1 then v else arbitrary)→ s2"
apply (case_tac "s", case_tac "a = None")
by (auto intro!: eval.Cond)

lemma IfI: "⟦G⊢s -e-≻v→ s1; G⊢s1 -(if the_Bool v then c1 else c2)→
s2⟧
                ⟹ G⊢s -If(e) c1 Else c2→ s2"
apply (case_tac "s", case_tac "a = None")
by (auto intro!: eval.If)

lemma MethdI: "G⊢s -body G C sig-≻v→ s' ⟹ G⊢s -Methd C sig-≻v→
s'"
apply (case_tac "s", case_tac "a = None")
by (auto intro!: eval.Methd)

lemma eval_Call: "⟦G⊢Norm s0 -e-≻a'→ s1; G⊢s1 -ps≐≻pvs→ s2;
       C = target mode (snd s2) a' cT;
       G⊢init_lvars G C (mn,pTs) mode a' pvs s2 -Methd C (mn,pTs)-≻
v→ s3;
       s3' = restore_lvars s2 s3⟧ ⟹
       G⊢Norm s0 -{t,cT,mode}e..mn({pTs}ps)-≻v→ s3'"
apply (drule eval.Call, assumption)
```

```
apply (rule HOL.refl)
apply simp+
done

lemma eval_Init: "⟦if inited C (globs s0) then s3 = Norm s0 else
      G⊢Norm (init_class_obj G C s0)
         −(if C = Object then Skip else init (fst (the (class G C))))→
s1 ∧
      G⊢set_lvars empty s1 −(snd (snd (snd (snd (the (class G C))))))→
s2 ∧
      s3 = restore_lvars s1 s2⟧ ⟹
  G⊢Norm s0 −init C→ s3"
apply (rule surjective_pairing5 [THEN eval.Init])
apply auto
done


lemma init_done: "initd C s ⟹ G⊢s −init C→ s"
apply (case_tac "s", simp)
apply (case_tac "a")
apply  safe
apply (rule eval_Init)
apply   auto
done

lemma eval_StatRef:
"G⊢s −StatRef rt-≻(if fst s=None then Null else arbitrary)→ s"
apply (case_tac "s", simp)
apply (case_tac "a = None")
apply (auto del: eval.Xcpt intro!: eval.intros)
done


lemma SkipD [dest!]: "G⊢s −Skip→ s' ⟹ s' = s"
apply (erule eval_cases)
by auto

lemma Skip_eq [simp]: "G⊢s −Skip→ s' = (s = s')"
by auto


lemma init_retains_locals [rule_format (no_asm)]: "G⊢s −t≻→ (w,s')
⟹
  (∀C. t=In1r (init C) ⟶ locals (snd s) = locals (snd s'))"
apply (erule eval.induct)
apply (simp (no_asm_use) split del: split_if_asm option.split_asm)+
apply auto
done
```

lemma *halloc_xcpt [dest!]:*
"⋀s'. G⊢(Some xc,s) −halloc oi≻a→ s' ⟹ s'=(Some xc,s)"
apply *(erule_tac halloc_elim_cases)*
by *auto*

lemma *eval_Methd:*
"G⊢s −In1l(body G C sig)≻→ (w,s') ⟹ G⊢s −In1l(Methd C sig)≻→ (w,s')"
apply *(case_tac "s")*
apply *(case_tac "a")*
apply *clarsimp+*
apply *(erule eval.Methd)*
apply *(drule eval_xcpt_lemma)*
apply *force*
done

43 single$_v$alued

lemma *unique_halloc [rule_format (no_asm)]:*
"⋀s as as'. (s,oi,as)∈halloc G ⟹ (s,oi,as')∈halloc G ⟶ as'=as"
apply *(simp (no_asm_simp) only: split_tupled_all)*
apply *(erule halloc.induct)*
apply *(auto elim!: halloc_elim_cases split del: split_if split_if_asm)*
apply *(drule trans [THEN sym], erule sym)*
defer
apply *(drule trans [THEN sym], erule sym)*
apply *auto*
done

lemma *single_valued_halloc:*
"single_valued {((s,oi),(a,s')). G⊢s −halloc oi≻a → s'}"
apply *(unfold single_valued_def)*
by *(clarsimp, drule (1) unique_halloc, auto)*

lemma *unique_sxalloc [rule_format (no_asm)]:*
"⋀s s'. G⊢s −sxalloc→ s' ⟹ G⊢s −sxalloc→ s'' ⟶ s'' = s'"
apply *(simp (no_asm_simp) only: split_tupled_all)*
apply *(erule sxalloc.induct)*
apply *(auto dest: unique_halloc elim!: sxalloc_elim_cases*
split del: split_if split_if_asm)
done

lemma *single_valued_sxalloc: "single_valued {(s,s'). G⊢s −sxalloc→ s'}"*
apply *(unfold single_valued_def)*

```
apply (blast dest: unique_sxalloc)
done

lemma split_pairD: "(x,y) = p ⟹ x = fst p & y = snd p"
by auto

lemma unique_eval [rule_format (no_asm)]:
  "G⊢s −t≻→ ws ⟹ (∀ws'. G⊢s −t≻→ ws' ⟶ ws' = ws)"
apply (case_tac "ws")
apply (simp only:)
apply (erule thin_rl)
apply (erule eval_induct)
apply (tactic {* ALLGOALS (EVERY'
      [strip_tac, rotate_tac ~1, eresolve_tac (thms "eval_elim_cases")])
*})

prefer 24
apply (simp (no_asm_use) only: split add: split_if_asm)
prefer 26
apply (drule (1) trans [OF sym])
apply (case_tac "inited C (globs s0)", (simp only: if_True if_False)+)
prefer 22
apply (simp (no_asm_use) only: split add: split_if_asm)
apply (blast dest: unique_sxalloc unique_halloc split_pairD)+
done


lemma single_valued_eval:
 "single_valued {((s,t),vs'). G⊢s −t≻→ vs'}"
apply (unfold single_valued_def)
by (clarify, drule (1) unique_eval, auto)

end



theory Evaln = Eval:

consts

  evaln :: "prog ⇒ (state × term × nat × vals × state) set"

syntax

  evaln :: "[prog, state, term,        nat, vals * state] => bool"
                                ("_|-_ -_>-_-> _"   [61,61,80,   61,61]
60)
  evarn :: "[prog, state, var  , vvar        , nat, state] => bool"
                                ("_|-_ -_=>_-_-> _" [61,61,90,61,61,61]
```

```
60)
  eval_n:: "[prog, state, expr , val         , nat, state] => bool"
                                ("_|-_ -_->_-_-> _" [61,61,80,61,61,61]
60)
  evalsn:: "[prog, state, expr list, val list, nat, state] => bool"
                                ("_|-_ -_#>_-_-> _" [61,61,61,61,61,61]
60)
  execn :: "[prog, state, stmt ,               nat, state] => bool"
                                ("_|-_ -_-_-> _"    [61,61,65,   61,61]
60)
```

syntax *(xsymbols)*

```
  evaln :: "[prog, state, term,         nat, vals × state] ⇒ bool"
                                ("_⊢_ -_≻-_→ _"   [61,61,80,   61,61]
60)
  evarn :: "[prog, state, var  , vvar         , nat, state] ⇒ bool"
                                ("_⊢_ -_=≻_-_→ _" [61,61,90,61,61,61]
60)
  eval_n:: "[prog, state, expr , val ,          nat, state] ⇒ bool"
                                ("_⊢_ -_-≻_-_→ _" [61,61,80,61,61,61]
60)
  evalsn:: "[prog, state, expr list, val  list, nat, state] ⇒ bool"
                                ("_⊢_ -_≐≻_-_→ _" [61,61,61,61,61,61]
60)
  execn :: "[prog, state, stmt ,                nat, state] ⇒ bool"
                                ("_⊢_ -_-_→ _"      [61,61,65,   61,61]
60)
```

translations

```
  "G⊢s -t    ≻-n→  w___s' " == "(s,t,n,w___s') ∈ evaln G"
  "G⊢s -t    ≻-n→ (w,  s')" <= "(s,t,n,w,  s') ∈ evaln G"
  "G⊢s -t    ≻-n→ (w,x,s')" <= "(s,t,n,w,x,s') ∈ evaln G"
  "G⊢s -c     -n→ (x,s')" <= "G⊢s -In1r  c≻-n→ (•    ,x,s')"
  "G⊢s -c     -n→    s' " == "G⊢s -In1r  c≻-n→ (•    ,  s')"
  "G⊢s -e-≻v  -n→ (x,s')" <= "G⊢s -In1l e≻-n→ (In1 v ,x,s')"
  "G⊢s -e-≻v  -n→    s' " == "G⊢s -In1l e≻-n→ (In1 v ,  s')"
  "G⊢s -e=≻vf -n→ (x,s')" <= "G⊢s -In2  e≻-n→ (In2 vf,x,s')"
  "G⊢s -e=≻vf -n→    s' " == "G⊢s -In2  e≻-n→ (In2 vf,  s')"
  "G⊢s -e≐≻v  -n→ (x,s')" <= "G⊢s -In3  e≻-n→ (In3 v ,x,s')"
  "G⊢s -e≐≻v  -n→    s' " == "G⊢s -In3  e≻-n→ (In3 v ,  s')"
```

inductive *"evaln G"* **intros**

```
  Xcpt: "G⊢(Some xc,s) -t≻-n→ (arbitrary3 t,(Some xc,s))"
```

```
  LVar: "G⊢Norm s -LVar vn=≻lvar vn s-n→ Norm s"

  FVar: "⟦G⊢Norm s0 -init C-n→ s1; G⊢s1 -e-≻a'-n→ s2;
          (v,s2') = fvar C stat fn a' s2⟧ ⟹
          G⊢Norm s0 -{C,stat}e..fn=≻v-n→ s2'"

  AVar: "⟦G⊢ Norm s0 -e1-≻a-n→ s1 ; G⊢s1 -e2-≻i-n→ s2;
          (v,s2') = avar G i a s2⟧ ⟹
                      G⊢Norm s0 -e1.[e2]=≻v-n→ s2'"


  NewC: "⟦G⊢Norm s0 -init C-n→ s1;
          G⊢     s1 -halloc (CInst C)≻a→ s2⟧ ⟹
                                  G⊢Norm s0 -NewC C-≻Addr a-n→ s2"

  NewA: "⟦G⊢Norm s0 -init_comp_ty T-n→ s1; G⊢s1 -e-≻i'-n→ s2;
          G⊢xupd (check_neg i') s2 -halloc (Arr T (the_Intg i'))≻a→
s3⟧ ⟹
                                G⊢Norm s0 -New T[e]-≻Addr a-n→ s3"

  Cast: "⟦G⊢Norm s0 -e-≻v-n→ s1;
          s2 = xupd (raise_if (¬G,snd s1⊢v fits T) ClassCast) s1⟧ ⟹
                                G⊢Norm s0 -Cast T e-≻v-n→ s2"

  Inst: "⟦G⊢Norm s0 -e-≻v-n→ s1;
          b = (v≠Null ∧ G,snd s1⊢v fits RefT T)⟧ ⟹
                              G⊢Norm s0 -e InstOf T-≻Bool b-n→ s1"

  Lit:                      "G⊢Norm s -Lit v-≻v-n→ Norm s"

  Super:                    "G⊢Norm s -Super-≻val_this s-n→ Norm s"

  Acc:  "⟦G⊢Norm s0 -va=≻(v,f)-n→ s1⟧ ⟹
                                  G⊢Norm s0 -Acc va-≻v-n→ s1"

  Ass:  "⟦G⊢Norm s0 -va=≻(w,f)-n→ s1;
          G⊢     s1 -e-≻v     -n→ s2⟧ ⟹
                                   G⊢Norm s0 -va:=e-≻v-n→ assign f
v s2"

  Cond: "⟦G⊢Norm s0 -e0-≻b-n→ s1;
```

```
        G⊢     s1 −(if the_Bool b then e1 else e2)-≻v−n→ s2⟧ ⟹
                          G⊢Norm s0 −e0 ? e1 : e2-≻v−n→ s2"

  Call: "⟦G⊢Norm s0 −e-≻a'−n→ s1; G⊢s1 −args≐≻vs−n→ s2;
         C = target mode (snd s2) a' cT;
         G⊢init_lvars G C (mn,pTs) mode a' vs s2 −Methd C (mn,pTs)-≻v−n→
s3⟧
      ⟹ G⊢Norm s0 −{t,cT,mode}e..mn({pTs}args)-≻v−n→ (restore_lvars
s2 s3)"

  Methd:"⟦G⊢Norm s0 −body G C sig-≻v−n→ s1⟧ ⟹
                               G⊢Norm s0 −Methd C sig-≻v−Suc n→ s1"

  Body: "⟦G⊢Norm s0−init D−n→ s1; G⊢s1 −c−n→ s2; G⊢s2 −e-≻v−n→
s3⟧⟹
                               G⊢Norm s0 −Body D c e-≻v−n→ s3"



  Nil:
                               "G⊢Norm s0 −[]≐≻[]−n→ Norm s0"

  Cons: "⟦G⊢Norm s0 −e -≻ v −n→ s1;
         G⊢     s1 −es≐≻vs−n→ s2⟧ ⟹
                           G⊢Norm s0 −e#es≐≻v#vs−n→ s2"



  Skip:                              "G⊢Norm s −Skip−n→ Norm s"

  Expr: "⟦G⊢Norm s0 −e-≻v−n→ s1⟧ ⟹
                                G⊢Norm s0 −Expr e−n→ s1"

  Comp: "⟦G⊢Norm s0 −c1 −n→ s1;
         G⊢     s1 −c2 −n→ s2⟧ ⟹
                               G⊢Norm s0 −c1;; c2−n→ s2"

  If:   "⟦G⊢Norm s0 −e-≻b−n→ s1;
         G⊢     s1−(if the_Bool b then c1 else c2)−n→ s2⟧ ⟹
                     G⊢Norm s0 −If(e) c1 Else c2 −n→ s2"

  Loop: "⟦G⊢Norm s0 −e-≻b−n→ s1;
         if the_Bool b then (G⊢s1 −c−n→ s2 ∧ G⊢s2 −While(e) c−n→
s3)
                        else s3 = s1⟧ ⟹
                              G⊢Norm s0 −While(e) c−n→ s3"

  Throw:"⟦G⊢Norm s0 −e-≻a'−n→ s1⟧ ⟹
```

```
                                    G⊢Norm s0 −Throw e−n→ xupd (throw
a') s1"

  Try:  "⟦G⊢Norm s0 −c1−n→ s1; G⊢s1 −sxalloc→ s2;
          if G,s2⊢catch tn then G⊢new_xcpt_var vn s2 −c2−n→ s3 else
s3 = s2⟧⟹
                  G⊢Norm s0 −Try c1 Catch(tn vn) c2−n→ s3"

  Fin:  "⟦G⊢Norm s0 −c1−n→ (x1,s1);
          G⊢Norm s1 −c2−n→ s2⟧ ⟹
                   G⊢Norm s0 −c1 Finally c2−n→ xupd (xcpt_if (x1≠None)
x1) s2"

  Init: "⟦the (class G C) = (sc,si,fs,ms,ini);
          if inited C (globs s0) then s3 = Norm s0
          else (G⊢Norm (init_class_obj G C s0)
                  −(if C = Object then Skip else init sc)−n→ s1 ∧
                G⊢set_lvars empty s1 −ini−n→ s2 ∧ s3 = restore_lvars
s1 s2)⟧⟹
                 G⊢Norm s0 −init C−n→ s3"
monos
  if_def2

lemma evaln_eval: "⋀ws. G⊢s −t≻−n→ ws ⟹ G⊢s −t≻→ ws"
apply (simp (no_asm_simp) only: split_tupled_all)
apply (erule evaln.induct)
apply (tactic {* ALLGOALS (resolve_tac (thms "eval.intros")
   THEN_ALL_NEW TRY o atac) *})

apply (auto split del: split_if)
done

lemma Suc_le_D_lemma: "⟦Suc n <= m'; (⋀m. n <= m ⟹ P (Suc m)) ⟧ ⟹
P m'"
apply (frule Suc_le_D)
apply fast
done

lemma evaln_nonstrict [rule_format (no_asm), elim]:
  "⋀ws. G⊢s −t≻−n→ ws ⟹ ∀m. n≤m ⟶ G⊢s −t≻−m→ ws"
apply (simp (no_asm_simp) only: split_tupled_all)
apply (erule evaln.induct)
apply (tactic {* ALLGOALS (EVERY'[strip_tac, TRY o etac (thm "Suc_le_D_lemma"),
  REPEAT o smp_tac 1,
  resolve_tac (thms "evaln.intros") THEN_ALL_NEW TRY o atac]) *})

apply (auto split del: split_if)
done
```

```
lemmas evaln_nonstrict_Suc = evaln_nonstrict [OF _ le_refl [THEN le_SucI]]

lemma evaln_max2: "⟦G⊢s1 −t1≻−n1→ ws1; G⊢s2 −t2≻−n2→ ws2⟧ ⟹

            G⊢s1 −t1≻−max n1 n2→ ws1 ∧ G⊢s2 −t2≻−max n1 n2→ ws2"
apply (fast intro: le_maxI1 le_maxI2)
done

lemma evaln_max3: "⟦G⊢s1 −t1≻−n1→ ws1; G⊢s2 −t2≻−n2→ ws2; G⊢s3
−t3≻−n3→ ws3⟧ ⟹
 G⊢s1 −t1≻−max (max n1 n2) n3→ ws1 ∧
 G⊢s2 −t2≻−max (max n1 n2) n3→ ws2 ∧
 G⊢s3 −t3≻−max (max n1 n2) n3→ ws3"
apply (drule (1) evaln_max2, erule thin_rl)
apply (fast intro!: le_maxI1 le_maxI2)
done

lemma eval_evaln: "⋀ws. G⊢s −t≻→ ws ⟹ (∃n. G⊢s −t≻−n→ ws)"
apply (simp (no_asm_simp) only: split_tupled_all)
apply (erule eval.induct)
apply (tactic {* ALLGOALS
        (asm_full_simp_tac (HOL_basic_ss addsplits [split_if_asm])) *})
apply (tactic {* ALLGOALS (EVERY'[
   REPEAT o eresolve_tac [exE, conjE], rtac exI,
                    TRY o datac (thm "evaln_max3") 2, REPEAT o etac conjE,
  resolve_tac (thms "evaln.intros") THEN_ALL_NEW
  force_tac (HOL_cs, HOL_ss)]) *})
done

declare split_if     [split del] split_if_asm     [split del]
       option.split [split del] option.split_asm [split del]
inductive_cases evaln_cases: "G⊢s −t≻−n→ vs'"

inductive_cases evaln_elim_cases:
        "G⊢(Some xc, s) −t≻−n→  vs'"
        "G⊢Norm s −In1r Skip                    ≻−n→ xs'"
        "G⊢Norm s −In3  ([])                    ≻−n→ vs'"
        "G⊢Norm s −In3  (e#es)                  ≻−n→ vs'"
        "G⊢Norm s −In1l (Lit w)                 ≻−n→ vs'"
        "G⊢Norm s −In2  (LVar vn)               ≻−n→ vs'"
        "G⊢Norm s −In1l (Cast T e)              ≻−n→ vs'"
        "G⊢Norm s −In1l (e InstOf T)            ≻−n→ vs'"
        "G⊢Norm s −In1l (Super)                 ≻−n→ vs'"
        "G⊢Norm s −In1l (Acc va)                ≻−n→ vs'"
        "G⊢Norm s −In1r (Expr e)                ≻−n→ xs'"
        "G⊢Norm s −In1r (c1;; c2)               ≻−n→ xs'"
        "G⊢Norm s −In1l (Methd C sig)           ≻−n→ xs'"
        "G⊢Norm s −In1l (Body D c e)            ≻−n→ xs'"
        "G⊢Norm s −In1l (e0 ? e1 : e2)          ≻−n→ vs'"
```

```
        "G⊢Norm s -In1r (If(e) c1 Else c2)       ≻-n→ xs'"
        "G⊢Norm s -In1r (While(e) c)             ≻-n→ xs'"
        "G⊢Norm s -In1r (c1 Finally c2)          ≻-n→ xs'"
        "G⊢Norm s -In1r (Throw e)                ≻-n→ xs'"
        "G⊢Norm s -In1l (NewC C)                 ≻-n→ vs'"
        "G⊢Norm s -In1l (New T[e])               ≻-n→ vs'"
        "G⊢Norm s -In1l (Ass va e)               ≻-n→ vs'"
        "G⊢Norm s -In1r (Try c1 Catch(tn vn) c2) ≻-n→ xs'"
        "G⊢Norm s -In2  ({C,stat}e..fn)          ≻-n→ vs'"
        "G⊢Norm s -In2  (e1.[e2])                ≻-n→ vs'"
        "G⊢Norm s -In1l ({t,cT,mode}e..mn({pT}p)) ≻-n→ vs'"
        "G⊢Norm s -In1r (init C)                 ≻-n→ xs'"
declare split_if     [split] split_if_asm     [split]
        option.split [split] option.split_asm [split]

lemma evaln_Inj_elim: "G⊢s -t≻-n→ (w,s') ⟹ case t of In1 ec ⇒

  (case ec of Inl e ⇒ (∃v. w = In1 v) | Inr c ⇒ w = •)
  | In2 e ⇒ (∃v. w = In2 v) | In3 e ⇒ (∃v. w = In3 v)"
apply (erule evaln_cases , auto)
apply (induct_tac "t")
apply   (induct_tac "a")
apply auto
done

ML_setup {*
fun enf nam inj rhs =
let
  val name = "evaln_" ^ nam ^ "_eq"
  val lhs = "G⊢s -" ^ inj ^ " t≻-n→ (w, s')"
  val () = qed_goal name (the_context()) (lhs ^ " = (" ^ rhs ^ ")")
        (K [Auto_tac, ALLGOALS (ftac (thm "evaln_Inj_elim")) THEN Auto_tac])
  fun is_Inj (Const (inj,_) $ _) = true
    | is_Inj _                   = false
  fun pred (_ $ (Const ("Pair",_) $ _ $ (Const ("Pair", _) $ _ $
    (Const ("Pair", _) $ _ $ (Const ("Pair", _) $ x $ _ )))) $ _ ) = is_Inj
x
in
  make_simproc name lhs pred (thm name)
end;

val evaln_expr_proc = enf "expr" "In1l" "∃v.  w=In1 v  ∧ G⊢s -t-≻v
-n→ s'";
val evaln_var_proc  = enf "var"  "In2"  "∃vf. w=In2 vf ∧ G⊢s -t=≻vf-n→
s'";
val evaln_exprs_proc= enf "exprs""In3"  "∃vs. w=In3 vs ∧ G⊢s -t≐≻vs-n→
s'";
val evaln_stmt_proc = enf "stmt" "In1r" "     w=•      ∧ G⊢s -t     -n→
s'";
```

```
Addsimprocs [evaln_expr_proc,evaln_var_proc,evaln_exprs_proc,evaln_stmt_proc];

bind_thms ("evaln_XcptIs", sum3_instantiate (thm "evaln.Xcpt"))
*}
declare evaln_XcptIs [intro!]

lemma evaln_xcpt_lemma: "G⊢s -e≻-n→ (v,s') ⟹
 fst s = Some xc ⟶ s' = s ∧ v = arbitrary3 e"
apply (erule evaln_cases , auto)
done

lemma evaln_xcpt: "⋀s'. G⊢(Some xc,s) -e≻-n→ (w,s') = (s' = (Some
xc,s) ∧
  w=arbitrary3 e ∧ G⊢(Some xc,s) -e≻-n→ (arbitrary3 e,(Some xc,s)))"
apply auto
apply (frule evaln_xcpt_lemma, auto)+
done

ML {*
local
  fun is_Some (Const ("Pair",_) $ (Const ("Option.option.Some",_) $ _)$
_) =true
    | is_Some _ = false
  fun pred (_ $ (Const ("Pair",_) $
     _ $ (Const ("Pair", _) $ _ $ (Const ("Pair", _) $ _ $
       (Const ("Pair", _) $ _ $ x)))) $ _ ) = is_Some x
in
  val evaln_xcpt_proc =
 make_simproc "evaln_xcpt" "G⊢(Some xc,s) -e≻-n→ (w,s')" pred (thm
"evaln_xcpt")
end;
Addsimprocs [evaln_xcpt_proc]
*}

lemma evaln_LitI: "G⊢s -Lit v-≻(if normal s then v else arbitrary)-n→
s"
apply (case_tac "s", case_tac "a = None")
by (auto intro!: evaln.Lit)

lemma CondI:
 "⋀s1. ⟦G⊢s -e-≻b-n→ s1; G⊢s1 -(if the_Bool b then e1 else e2)-≻v-n→
s2⟧ ⟹
  G⊢s -e ? e1 : e2-≻(if normal s1 then v else arbitrary)-n→ s2"
apply (case_tac "s", case_tac "a = None")
by (auto intro!: evaln.Cond)

lemma evaln_SkipI [intro!]: "G⊢s -Skip-n→ s"
apply (case_tac "s", case_tac "a = None")
by (auto intro!: evaln.Skip)
```

lemma *evaln_ExprI: "G⊢s −e-≻v−n→ s' ⟹ G⊢s −Expr e−n→ s'"*
apply *(case_tac "s", case_tac "a = None")*
by *(auto intro!: evaln.Expr)*

lemma *evaln_CompI: "⟦G⊢s −c1−n→ s1; G⊢s1 −c2−n→ s2⟧ ⟹ G⊢s −c1;; c2−n→ s2"*
apply *(case_tac "s", case_tac "a = None")*
by *(auto intro!: evaln.Comp)*

lemma *evaln_IfI:*
"⟦G⊢s −e-≻v−n→ s1; G⊢s1 −(if the_Bool v then c1 else c2)−n→ s2⟧ ⟹ G⊢s −If(e) c1 Else c2−n→ s2"
apply *(case_tac "s", case_tac "a = None")*
by *(auto intro!: evaln.If)*

lemma *evaln_SkipD [dest!]: "G⊢s −Skip−n→ s' ⟹ s' = s"*
by *(erule evaln_cases, auto)*

lemma *evaln_Skip_eq [simp]: "G⊢s −Skip−n→ s' = (s = s')"*
apply *auto*
done

end

theory *Example = Eval + WellForm:*

declare *widen.null [intro]*

lemma *wf_fdecl_def2: "⋀fd. wf_fdecl G fd = is_type G (snd (snd fd))"*
apply *(unfold wf_fdecl_def)*
apply *(simp (no_asm))*
done

declare *wf_fdecl_def2 [iff]*

44 type and expression names

datatype *tnam_ = HasFoo_ | Base_ | Ext_*
datatype *enam_ = arr_ | vee_ | z_ | e_*

consts

tnam_ :: "tnam_ ⇒ tnam"
enam_ :: "enam_ ⇒ ename"

axioms

```
inj_tnam_ [simp]: "(tnam_ x = tnam_ y) = (x = y)"
inj_enam_ [simp]: "(enam_ x = enam_ y) = (x = y)"

surj_tnam_: "∃m. n = tnam_ m"
surj_enam_: "∃m. n = enam_ m"
```

syntax

```
HasFoo :: tname
Base   :: tname
Ext    :: tname
arr :: ename
vee :: ename
z   :: ename
e   :: ename
```

translations

```
"HasFoo" == "TName (tnam_ HasFoo_)"
"Base"   == "TName (tnam_ Base_)"
"Ext"    == "TName (tnam_ Ext_)"
"arr"    ==        "enam_ arr_"
"vee"    ==        "enam_ vee_"
"z"      ==        "enam_ z_"
"e"      ==        "enam_ e_"
```

45 classes and interfaces

defs

```
Object_mdecls_def: "Object_mdecls ≡ []"
SXcpt_mdecls_def:  "SXcpt_mdecls  ≡ []"
```

consts

```
foo    :: mname
```

constdefs

```
foo_sig   :: sig
"foo_sig   ≡ (foo,[Class Base])"

foo_mhead :: mhead
"foo_mhead ≡ (False,[z],Class Base)"
```

constdefs

```
 Base_foo :: mdecl
"Base_foo ≡ (foo_sig, (foo_mhead,([],Skip,!!z)))"

 Ext_foo  :: mdecl
"Ext_foo  ≡ (foo_sig, ((False,[z],Class Ext),
            ([],Expr({Ext,False}Cast (Class Ext) (!!z)..vee :=
                    Lit (Intg #1)),Lit Null)
            ))"
```

constdefs

```
 arr_viewed_from :: "tname ⇒ var"
"arr_viewed_from C ≡ {Base,True}StatRef (ClassT C)..arr"

 BaseCl :: class
"BaseCl ≡ (Object, [HasFoo],
           [(arr, (True,  PrimT Boolean.[])),
            (vee, (False, Iface HasFoo    ))],
           [Base_foo],
           Expr(arr_viewed_from Base := New (PrimT Boolean)[Lit (Intg
#2)]))"

 ExtCl  :: class
"ExtCl  ≡ (Base  , [],
           [(vee, (False, PrimT Integer  ))],
           [Ext_foo],
           Skip)"
```

constdefs

```
 HasFooInt :: iface
"HasFooInt ≡ ([], [(foo_sig, foo_mhead)])"

 ifaces ::"idecl list"
"ifaces ≡ [(HasFoo,HasFooInt)]"

"classes" ::"cdecl list"
"classes ≡ [(Base,BaseCl),(Ext,ExtCl)]@standard_classes"
```

lemmas *table_classes_defs =*
classes_def standard_classes_def ObjectC_def SXcptC_def

lemma *table_ifaces [simp]: "table_of ifaces = empty(HasFoo↦HasFooInt)"*
apply *(unfold ifaces_def)*
apply *(simp (no_asm))*
done

lemma *table_classes_Object [simp]:*

```
 "table_of classes Object = Some (arbitrary, [], [], Object_mdecls, Skip)"
apply (unfold table_classes_defs)
apply (simp (no_asm))
done

lemma table_classes_SXcpt [simp]:
  "table_of classes (SXcpt xn) = Some (if xn = Throwable then Object
  else SXcpt Throwable, [], [], SXcpt_mdecls, Skip)"
apply (unfold table_classes_defs)
apply (induct_tac xn)
apply simp+
done

lemma table_classes_HasFoo [simp]: "table_of classes HasFoo = None"
apply (unfold table_classes_defs)
apply (simp (no_asm))
done

lemma table_classes_Base [simp]: "table_of classes Base = Some BaseCl"
apply (unfold table_classes_defs )
apply (simp (no_asm))
done
lemma table_classes_Ext [simp]: "table_of classes Ext = Some ExtCl"
apply (unfold table_classes_defs )
apply (simp (no_asm))
done
```

46 program

```
syntax
  tprg :: prog
translations
  "tprg" == "(ifaces,classes)"

constdefs
  test    :: "(ty)list ⇒ stmt"
 "test pTs ≡ e:==NewC Ext;;
             Try Expr({ClassT Base,ClassT Base,IntVir}!!e..
                    foo({pTs}[Lit Null]))
             Catch((SXcpt NullPointer) z)
              (While(Acc (Acc (arr_viewed_from Ext).[Lit (Intg #2)]))
Skip)"
```

47 well-structuredness

```
lemma not_Object_subcls_any [elim!]: "(Object, C) ∈ (subcls1 tprg)^+
⟹ R"
apply (auto dest!: tranclD subcls1D)
```

done

lemma *not_Throwable_subcls_SXcpt [elim!]:*
"(SXcpt Throwable, SXcpt xn) ∈ (subcls1 tprg)^+ ⟹ R"
apply *(auto dest!: tranclD subcls1D)*
done

lemma *not_SXcpt_n_subcls_SXcpt_n [elim!]:*
"(SXcpt xn, SXcpt xn) ∈ (subcls1 tprg)^+ ⟹ R"
apply *(auto dest!: tranclD subcls1D)*
apply *(drule rtranclD)*
apply *auto*
done

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

lemma *not_TName_n_subcls_TName_n [rule_format (no_asm), elim!]:*
"(TName tn, TName tn) ∈ (subcls1 tprg)^+ ⟶ R"
apply *(rule_tac n1 = "tn"* **in** *surj_tnam_ [THEN exE])*
apply *(erule ssubst)*
apply *(rule tnam_.induct)*
apply *safe*
apply *(auto dest!: tranclD subcls1D simp add: BaseCl_def ExtCl_def)*
apply *(drule rtranclD)*
apply *auto*
done

lemma *ws_idecl_HasFoo: "ws_idecl tprg HasFoo []"*
apply *(unfold ws_idecl_def)*
apply *(simp (no_asm))*
done

lemma *ws_cdecl_Object: "ws_cdecl tprg Object any"*
apply *(unfold ws_cdecl_def)*
apply *auto*
done

lemma *ws_cdecl_Throwable: "ws_cdecl tprg (SXcpt Throwable) Object"*
apply *(unfold ws_cdecl_def)*
apply *auto*
done

lemma *ws_cdecl_SXcpt: "ws_cdecl tprg (SXcpt xn) (SXcpt Throwable)"*
apply *(unfold ws_cdecl_def)*
apply *auto*

done

lemma *ws_cdecl_Base: "ws_cdecl tprg Base Object"*
apply *(unfold ws_cdecl_def)*
apply *auto*
done

lemma *ws_cdecl_Ext: "ws_cdecl tprg Ext Base"*
apply *(unfold ws_cdecl_def)*
apply *auto*
done

lemmas *ws_cdecls = ws_cdecl_SXcpt ws_cdecl_Object ws_cdecl_Throwable*
ws_cdecl_Base ws_cdecl_Ext

declare *not_Object_subcls_any [rule del]*
not_Throwable_subcls_SXcpt [rule del]
not_SXcpt_n_subcls_SXcpt_n [rule del]
not_Base_subcls_Ext [rule del] not_TName_n_subcls_TName_n [rule del]

lemma *ws_idecl_all: "G=tprg ⟹ (∀(I,(si,ib))∈set ifaces. ws_idecl G I si)"*
apply *(simp (no_asm) add: ifaces_def HasFooInt_def)*
apply *(auto intro!: ws_idecl_HasFoo)*
done

lemma *ws_cdecl_all: "G=tprg ⟹ (∀(C,(sc,cb))∈set classes. ws_cdecl G C sc)"*
apply *(simp (no_asm) add: classes_def BaseCl_def ExtCl_def)*
apply *(auto intro!: ws_cdecls simp add: standard_classes_def ObjectC_def*

SXcptC_def)
done

lemma *ws_tprg: "ws_prog tprg"*
apply *(unfold ws_prog_def)*
apply *(auto intro!: ws_idecl_all ws_cdecl_all)*
done

48 misc program properties (independent of well-structuredness)

lemma *single_iface [simp]: "is_iface tprg I = (I = HasFoo)"*
apply *(unfold ifaces_def)*
apply *(simp (no_asm))*
done

```
lemma empty_subint1 [simp]: "subint1 tprg = {}"
apply (unfold subint1_def ifaces_def HasFooInt_def)
apply auto
done

lemma unique_ifaces: "unique ifaces"
apply (unfold ifaces_def)
apply (simp (no_asm))
done

lemma unique_classes: "unique classes"
apply (unfold table_classes_defs )
apply simp
done

lemma SXcpt_subcls_Throwable [simp]: "tprg⊢SXcpt xn≼C SXcpt Throwable"
apply (rule SXcpt_subcls_Throwable_lemma)
apply force
done

lemma Ext_subcls_Base [simp]: "tprg⊢Ext ≼C Base"
apply (rule subcls_direct)
apply  (simp (no_asm) add: ExtCl_def)
apply (simp (no_asm))
done
```

49 fields and method lookup

```
lemma fields_tprg_Object [simp]: "fields tprg Object = []"
by (rule ws_tprg [THEN fields_emptyI], force+)

lemma fields_tprg_Throwable [simp]: "fields tprg (SXcpt Throwable) =
[]"
by (rule ws_tprg [THEN fields_emptyI], force+)

lemma fields_tprg_SXcpt [simp]: "fields tprg (SXcpt xn) = []"
apply (case_tac "xn = Throwable")
apply  (simp (no_asm_simp))
by (rule ws_tprg [THEN fields_emptyI], force+)

lemmas fields_rec_ = fields_rec [OF _ ws_tprg]

lemma fields_Base [simp]:
     "fields tprg Base = [((arr, Base), (True,  PrimT Boolean.[])),
                          ((vee, Base), (False, Iface HasFoo    ))]"
apply (subst fields_rec_)
apply   (auto simp add: BaseCl_def)
done
lemma fields_Ext [simp]:
```

```
 "fields tprg Ext  = [((vee, Ext ), (False, PrimT Integer))] @ fields
tprg Base"
apply (rule trans)
apply (rule fields_rec_)
apply   (auto simp add: ExtCl_def)
done

lemmas imethds_rec_ = imethds_rec [OF _ ws_tprg]
lemmas cmethd_rec_  = cmethd_rec  [OF _ ws_tprg]

lemma imethds_HasFoo [simp]:
  "imethds tprg HasFoo = o2s ∘ empty(foo_sig↦(HasFoo, foo_mhead))"
apply (rule trans)
apply (rule imethds_rec_)
apply  (auto simp add: HasFooInt_def)
done

lemma cmethd_tprg_Object [simp]: "cmethd tprg Object = empty"
apply (subst cmethd_rec_)
apply (auto simp add: Object_mdecls_def)
done

lemma cmethd_Base [simp]:
  "cmethd tprg Base = table_of [(λ(s,m). (s, Base, m)) Base_foo]"
apply (rule trans)
apply (rule cmethd_rec_)
apply   (auto simp add: BaseCl_def)
done

lemma cmethd_Ext [simp]: "cmethd tprg Ext = cmethd tprg Base ++
  table_of [(λ(s,m). (s, Ext, m)) Ext_foo]"
apply (rule trans)
apply (rule cmethd_rec_)
apply   (auto simp add: ExtCl_def)
done
```

50 well-formedness

```
lemma wf_HasFoo: "wf_idecl tprg (HasFoo, HasFooInt)"
apply (unfold wf_idecl_def HasFooInt_def)
apply (auto intro!: wf_mheadI ws_idecl_HasFoo
            simp add: foo_sig_def foo_mhead_def)
done

declare wt.Skip [rule del] wt.Init [rule del]
lemmas Base_foo_defs = Base_foo_def foo_sig_def foo_mhead_def
lemmas Ext_foo_defs  = Ext_foo_def  foo_sig_def
ML {* bind_thms ("wt_intros",map (rewrite_rule [id_def]) (thms "wt.intros"))
*}
```

```
lemmas wtIs = wt_Call wt_StatRef wt_intros

lemma wf_Base_foo: "wf_mdecl tprg Base Base_foo"
apply (unfold Base_foo_defs )
apply (auto intro!: wf_mdeclI wf_mheadI intro!: wtIs)
done

lemma wf_Ext_foo: "wf_mdecl tprg Ext Ext_foo"
apply (unfold Ext_foo_defs )
apply (auto intro!: wf_mdeclI wf_mheadI intro!: wtIs)
apply  (rule wt.Cast)
prefer 2
apply    simp
apply   (rule_tac [2] narrow.subcls [THEN cast.narrow])
apply   (unfold cfield_def)
apply   (auto intro!: wtIs)
done

lemma wf_BaseC: "wf_cdecl tprg (Base,BaseCl)"
apply (unfold wf_cdecl_def BaseCl_def arr_viewed_from_def)
apply (auto intro!: wf_Base_foo)
apply       (auto intro!: ws_cdecl_Base simp add: Base_foo_def foo_mhead_def)
apply (auto intro!: wtIs simp add: cfield_def)
done

lemma wf_ExtC: "wf_cdecl tprg (Ext,ExtCl)"
apply (unfold wf_cdecl_def ExtCl_def)
apply (auto intro!: wf_Ext_foo ws_cdecl_Ext)
apply (auto dest: map_of_SomeD
           simp add: Base_foo_defs Ext_foo_def hiding_entails_def)
done

lemma wf_idecl_all: "p=tprg ⟹ Ball (set ifaces) (wf_idecl p)"
apply (simp (no_asm) add: ifaces_def)
apply (simp (no_asm_simp))
apply (rule wf_HasFoo)
done

lemma wf_cdecl_all_standard_classes:
  "Ball (set standard_classes) (wf_cdecl tprg)"
apply (unfold standard_classes_def Let_def
       ObjectC_def SXcptC_def Object_mdecls_def SXcpt_mdecls_def)
apply (simp (no_asm) add: wf_cdecl_def ws_cdecls)
done

lemma wf_cdecl_all: "p=tprg ⟹ Ball (set classes) (wf_cdecl p)"
apply (simp (no_asm) add: classes_def)
apply (simp (no_asm_simp))
apply   (rule wf_BaseC [THEN conjI])
```

apply *(rule wf_ExtC [THEN conjI])*
apply *(rule wf_cdecl_all_standard_classes)*
done

theorem *wf_tprg: "wf_prog tprg"*
apply *(unfold wf_prog_def Let_def)*
apply *(simp (no_asm) add: unique_ifaces unique_classes)*
apply *(rule conjI)*
apply *((simp (no_asm) add: classes_def standard_classes_def))*
apply *(rule conjI)*
apply *(cut_tac xn_cases)*
apply *((simp (no_asm_simp) add: classes_def standard_classes_def))*
apply *(auto intro!: wf_idecl_all wf_cdecl_all)*
done

51 max$_s$*pec*

lemma *appl_methds_Base_foo:*
"appl_methds tprg (ClassT Base) (foo, [NT]) =
{((ClassT Base, (False,[z],Class Base)), [Class Base])}"
apply *(unfold appl_methds_def)*
apply *(simp (no_asm))*
apply *(subgoal_tac "tprg⊢NT⪯ Class Base")*
apply *(auto simp add: cmheads_def Base_foo_defs)*
done

lemma *max_spec_Base_foo: "max_spec tprg (ClassT Base) (foo, [NT]) =*
{((ClassT Base, (False,[z],Class Base)), [Class Base])}"
apply *(unfold max_spec_def)*
apply *(simp (no_asm) add: appl_methds_Base_foo)*
apply *auto*
done

52 well-typedness

lemma *wt_test: "(tprg, empty(EName e↦Class Base))⊢test ?pTs::√"*
apply *(unfold test_def arr_viewed_from_def)*

apply *(rule wtIs)*
apply *(rule wtIs)*
apply *(rule wtIs)*
apply *(rule wtIs)*
apply *(simp)*
apply *(simp)*
apply *(simp)*
apply *(rule wtIs)*
apply *(simp)*
apply *(simp)*

```
apply (rule wtIs )
prefer 4
apply    (simp)
defer
apply    (rule wtIs )
apply    (rule wtIs )
apply       (rule wtIs )
apply       (rule wtIs )
apply        (simp)
apply       (simp)
apply      (rule wtIs )
apply       (rule wtIs )
apply       (simp)
apply      (rule wtIs )
apply     (simp)
apply     (rule max_spec_Base_foo)
apply    (simp)
apply   (simp)
apply  (simp)
apply (rule wtIs )
apply  (rule wtIs )
apply   (rule wtIs )
apply   (rule wtIs )
apply   (rule wtIs )
apply    (rule wtIs )
apply    (simp)
apply   (simp add: cfield_def)
apply  (rule wtIs )
apply  (simp)
apply (rule wtIs )
done
```

53 execution

```
lemma alloc_one: "⋀a obj. ⟦the (new_Addr h) = a; atleast_free h (Suc
n)⟧ ⟹
  new_Addr h = Some a ∧ atleast_free (h(a↦obj)) n"
apply (frule atleast_free_SucD)
apply (drule atleast_free_Suc [THEN iffD1])
apply clarsimp
apply (frule new_Addr_SomeI)
apply force
done

declare fvar_def2 [simp] avar_def2 [simp] init_lvars_def2 [simp]
declare init_obj_def [simp] var_tys_def [simp] fields_table_def [simp]
declare BaseCl_def [simp] ExtCl_def [simp] Ext_foo_def [simp]
        Base_foo_defs  [simp]
```

ML *{* bind_thms ("eval_intros", map*
(simplify (simpset() delsimps [thm "Skip_eq"]
addsimps [thm "lvar_def"]) o
rewrite_rule [thm "assign_def",Let_def]) (thms "eval.intros"))
**}*
lemmas *eval_Is = eval_Init eval_StatRef XcptIs eval_intros*

consts
a :: loc
b :: loc
c :: loc

syntax

tprg :: prog

obj_a :: obj
obj_b :: obj
obj_c :: obj
arr_N :: "(vn, val) table"
arr_a :: "(vn, val) table"
globs1 :: globs
globs2 :: globs
globs3 :: globs
globs8 :: globs
locs3 :: locals
locs4 :: locals
locs8 :: locals
s0 :: state
s0' :: state
s9' :: state
s1 :: state
s1' :: state
s2 :: state
s2' :: state
s3 :: state
s3' :: state
s4 :: state
s4' :: state
s6' :: state
s7' :: state
s8 :: state
s8' :: state

translations

"tprg" == "(ifaces,classes)"

"obj_a" <= "(Arr (PrimT Boolean) #2, empty(Inr #0↦Bool False)(Inr

```
#1↦Bool False))"
  "obj_b"   <= "(CInst Ext,(empty(Inl (vee, Base)↦Null   )
                                (Inl (vee, Ext )↦Intg #0)))"
  "obj_c"   == "(CInst (SXcpt NullPointer),empty)"
  "arr_N"   == "empty(Inl (arr, Base)↦Null)"
  "arr_a"   == "empty(Inl (arr, Base)↦Addr a)"
  "globs1"  == "empty(Inr Ext   ↦(arbitrary, empty))
                      (Inr Base  ↦(arbitrary, arr_N))
                      (Inr Object↦(arbitrary, empty))"
  "globs2"  == "empty(Inr Ext   ↦(arbitrary, empty))
                      (Inr Object↦(arbitrary, empty))
                      (Inl a↦obj_a)
                      (Inr Base  ↦(arbitrary, arr_a))"
  "globs3"  == "globs2(Inl b↦obj_b)"
  "globs8"  == "globs3(Inl c↦obj_c)"
  "locs3"   == "empty(Inl e↦Addr b)"
  "locs4"   == "empty(Inl z↦Null)(Inr()↦Addr b)"
  "locs8"   == "locs3(Inl z↦Addr c)"
  "s0"  == "       st empty  empty"
  "s0'" == " Norm  s0"
  "s1"  == "       st globs1 empty"
  "s1'" == " Norm  s1"
  "s2"  == "       st globs2 empty"
  "s2'" == " Norm  s2"
  "s3"  == "       st globs3 locs3 "
  "s3'" == " Norm  s3"
  "s4"  == "       st globs3 locs4"
  "s4'" == " Norm  s4"
  "s6'" == "(Some (StdXcpt NullPointer), s4)"
  "s7'" == "(Some (StdXcpt NullPointer), s3)"
  "s8"  == "       st globs8 locs8"
  "s8'" == " Norm  s8"
  "s9'" == "(Some (StdXcpt IndOutBound), s8)"

declare Pair_eq [simp del]
lemma exec_test:
"⟦the (new_Addr (heap  s1)) = a;
  the (new_Addr (heap ?s2)) = b;
  the (new_Addr (heap ?s3)) = c⟧ ⟹
  atleast_free  (heap s0) 4 ⟹
  tprg⊢s0' −test [Class Base]→ ?s9'"
apply (unfold test_def arr_viewed_from_def)

apply (simp (no_asm_use))
apply (drule (1) alloc_one, clarsimp)
apply (rule eval_Is )
apply  (erule_tac V = "the (new_Addr ?h) = c" in thin_rl)
apply  (erule_tac [2] V = "new_Addr ?h = Some a" in thin_rl)
apply  (erule_tac [2] V = "atleast_free ?h 4" in thin_rl)
```

```
apply  (rule eval_Is )
apply  (rule eval_Is )
apply   (rule eval_Is )
apply  (rule eval_Is )

apply   (erule_tac V = "the (new_Addr ?h) = b" in thin_rl)
apply   (erule_tac V = "atleast_free ?h 3" in thin_rl)
apply   (erule_tac [2] V = "atleast_free ?h 4" in thin_rl)
apply   (erule_tac [2] V = "new_Addr ?h = Some a" in thin_rl)
apply   (rule eval_Is )
apply   (simp)
apply   (rule conjI)
prefer 2 apply (rule conjI HOL.refl)+
apply   (rule eval_Is )
apply   (simp add: arr_viewed_from_def)
apply   (rule conjI)
apply    (rule eval_Is )
apply    (simp)
apply    (rule conjI, rule HOL.refl)+
apply    (rule HOL.refl)
apply   (simp)
apply   (rule conjI, rule_tac [2] HOL.refl)
apply   (rule eval_Is )
apply   (rule eval_Is )
apply    (rule eval_Is )
apply      (rule init_done, simp)
apply     (rule eval_Is )
apply    (simp)
apply   (rule eval_Is )
apply     (simp)
apply    (rule eval_Is )
apply   (simp)
apply   (rule halloc.New)
apply    (simp (no_asm_simp))
apply   (drule atleast_free_weaken,rotate_tac -1,drule atleast_free_weaken)
apply   (simp (no_asm_simp))
apply  (simp add: upd_gobj_def)

apply  (rule halloc.New)
apply   (drule alloc_one)
prefer 2 apply fast
apply   (simp (no_asm_simp))
apply  (drule atleast_free_weaken)
apply  force
apply (simp)
apply (drule alloc_one)
apply  (simp (no_asm_simp))
apply clarsimp
apply (erule_tac V = "atleast_free ?h 3" in thin_rl)
```

```
apply (drule_tac x = "a" in new_AddrD2 [THEN spec])
apply (simp (no_asm_use))
apply (rule eval_Is )
apply   (rule eval_Is )

apply   (rule eval_Is )
apply      (rule eval_Is )
apply      (rule eval_Is )
apply     (rule eval_Is )
apply      (rule eval_Is )
apply     (rule eval_Is )
apply    (simp)
apply   (simp)
apply   (rule eval_Is )
apply   (simp add: body_def Let_def)
apply   (rule eval_Is )
apply     (rule init_done, simp)
apply    (rule eval_Is )
apply    (rule eval_Is )
apply     (rule eval_Is )
apply       (rule init_done, simp)
apply      (rule eval_Is )
apply       (rule eval_Is )
apply       (rule eval_Is )
apply      (simp)
apply     (simp split del: split_if)
apply    (rule eval_Is )
apply   (simp)
apply   (rule HOL.refl [THEN conjI])+
apply   (rule eval_Is )
apply  (simp)

apply  (rule sxalloc.intros)
apply  (rule halloc.New)
apply   (erule alloc_one [THEN conjunct1])
apply   (simp (no_asm_simp))
apply  (simp (no_asm_simp))
apply (simp add: gupd_def lupd_def obj_ty_def split del: split_if)
apply (drule alloc_one [THEN conjunct1])
apply  (simp (no_asm_simp))
apply (erule_tac V = "atleast_free ?h 2" in thin_rl)
apply (drule_tac x = "a" in new_AddrD2 [THEN spec])
apply simp
apply (rule eval_Is )
apply  (rule eval_Is )
apply  (rule eval_Is )
apply    (rule eval_Is )
apply    (rule eval_Is )
apply      (rule init_done, simp)
```

```
apply     (rule eval_Is )
apply    (simp)
apply   (rule eval_Is )
apply  (simp (no_asm_simp))
apply (simp add: in_bounds_def)
apply (tactic{* fast_tac (claset_of(theory "Main") addIs (thms "eval_Is"))
1 *})
done
declare Pair_eq [simp]

end



theory Conform = State:

types   env_ = "prog × (lname, ty) table"
```

54 extension of global store

constdefs

```
  gext    :: "st ⇒ st ⇒ bool"                  ("_≤|_"         [71,71]
70)
   "s≤|s' ≡ ∀r. ∀(oi, fs)∈globs s r: ∃(oi',fs')∈globs s' r: oi' =
oi"

lemma gext_objD: "⟦s≤|s'; globs s r = Some (oi, fs)⟧ ⟹ ∃fs'. globs
s' r = Some (oi, fs')"
apply (simp only: gext_def)
by force

lemma rev_gext_objD: "⟦globs s r = Some (oi, fs); s≤|s'⟧ ⟹ ∃fs'. globs
s' r = Some (oi, fs')"
by (auto elim: gext_objD)

lemma init_class_obj_inited:
   "init_class_obj G C s1≤|s2 ⟹ inited C (globs s2)"
apply (unfold inited_def init_obj_def)
apply (auto dest!: gext_objD)
done

lemma gext_refl [intro!, simp]: "s≤|s"
apply (unfold gext_def)
apply (fast del: fst_splitE)
done

lemma gext_gupd [simp, elim!]: "⋀s. globs s r = None ⟹ s≤|gupd(r↦x)s"
by (auto simp: gext_def)
```

lemma *gext_new [simp, elim!]: "⋀s. globs s r = None ⟹ s≤|init_obj G oi r s"*
apply *(simp only: init_obj_def)*
apply *(erule_tac gext_gupd)*
done

lemma *gext_trans [elim]: "⋀X. ⟦s≤|s'; s'≤|s''⟧ ⟹ s≤|s''"*
by *(force simp: gext_def)*

lemma *gext_upd_gobj [intro!]: "s≤|upd_gobj r n v s"*
apply *(simp only: gext_def)*
apply *auto*
apply *(case_tac "ra = r")*
apply *auto*
apply *(case_tac "globs s r = None")*
apply *auto*
done

lemma *gext_cong1 [simp]: "set_locals l s1≤|s2 = s1≤|s2"*
by *(auto simp: gext_def)*

lemma *gext_cong2 [simp]: "s1≤|set_locals l s2 = s1≤|s2"*
by *(auto simp: gext_def)*

lemma *gext_lupd1 [simp]: "lupd(vn↦v)s1≤|s2 = s1≤|s2"*
by *(auto simp: gext_def)*

lemma *gext_lupd2 [simp]: "s1≤|lupd(vn↦v)s2 = s1≤|s2"*
by *(auto simp: gext_def)*

lemma *inited_gext: "⟦inited C (globs s); s≤|s'⟧ ⟹ inited C (globs s')"*
apply *(unfold inited_def)*
apply *(auto dest: gext_objD)*
done

55 value conformance

constdefs

conf :: "prog ⇒ st ⇒ val ⇒ ty ⇒ bool" ("_,_⊢_::≼_" [71,71,71,71] 70)
"G,s⊢v::≼T ≡ ∃T'∈typeof (λa. option_map obj_ty (heap s a)) v:G⊢T'≼T"

lemma *conf_cong [simp]: "G,set_locals l s⊢v::≼T = G,s⊢v::≼T"*

```
by (auto simp: conf_def)

lemma conf_lupd [simp]: "G,lupd(vn↦va)s⊢v::≼T = G,s⊢v::≼T"
by (auto simp: conf_def)

lemma conf_PrimT [simp]: "∀dt. typeof dt v = Some (PrimT t) ⟹ G,s⊢v::≼PrimT
t"
apply (simp add: conf_def)
done

lemma conf_litval [rule_format (no_asm)]: "typeof (λa. None) v = Some
T ⟶ G,s⊢v::≼T"
apply (unfold conf_def)
apply (rule val.induct)
apply auto
done

lemma conf_Null [simp]: "G,s⊢Null::≼T = G⊢NT≼T"
by (simp add: conf_def)

lemma conf_Addr: "G,s⊢Addr a::≼T = (∃obj. heap s a = Some obj ∧ G⊢obj_ty
obj≼T)"
by (auto simp: conf_def)

lemma conf_AddrI:"⟦heap s a = Some obj; G⊢obj_ty obj≼T⟧ ⟹ G,s⊢Addr
a::≼T"
apply (rule conf_Addr [THEN iffD2])
by fast

lemma defval_conf [rule_format (no_asm), elim]:
  "is_type G T ⟶ G,s⊢default_val T::≼T"
apply (unfold conf_def)
apply (induct "T")
apply (auto intro: prim_ty.induct)
done

lemma conf_widen [rule_format (no_asm), elim]:
  "G⊢T≼T' ⟹ G,s⊢x::≼T ⟶ ws_prog G ⟶ G,s⊢x::≼T'"
apply (unfold conf_def)
apply (rule val.induct)
apply (auto elim: ws_widen_trans)
done

lemma conf_gext [rule_format (no_asm), elim]: "G,s⊢v::≼T ⟶ s≤|s' ⟶
G,s'⊢v::≼T"
apply (unfold gext_def conf_def)
apply (rule val.induct)
apply force+
done
```

lemma *conf_list_widen [rule_format (no_asm)]: "ws_prog G ⟹ ∀Ts Ts'. list_all2 (conf G s) vs Ts ⟶ G⊢Ts[≼] Ts' ⟶ list_all2 (conf G s) vs Ts'"*
apply *(unfold widens_def)*
apply *(rule list_all2_trans)*
apply *auto*
done

lemma *conf_RefTD [rule_format (no_asm)]: "G,s⊢a'::≼RefT T ⟶ a' = Null ∨ (∃a oi vs T'. a' = Addr a ∧ heap s a = Some (oi,vs) ∧ obj_ty (oi,vs) = T' ∧ G⊢T'≼RefT T)"*
apply *(unfold conf_def)*
apply *(induct_tac "a'")*
apply *(auto dest: widen_PrimT)*
done

56 value list conformance

constdefs

lconf :: "prog ⇒ st ⇒ ('a, val) table ⇒ ('a, ty) table ⇒ bool" ("_,_⊢_[::≼]_" [71,71,71,71] 70)
"G,s⊢vs[::≼]Ts ≡ ∀n. ∀T∈Ts n: ∃v∈vs n: G,s⊢v::≼T"

lemma *lconfD: "⟦G,s⊢vs[::≼]Ts; Ts n = Some T⟧ ⟹ G,s⊢(the (vs n))::≼T"*
by *(force simp: lconf_def)*

lemma *lconf_cong [simp]: "⋀s. G,set_locals x s⊢l[::≼]L = G,s⊢l[::≼]L"*
by *(auto simp: lconf_def)*

lemma *lconf_lupd [simp]: "G,lupd(vn↦v)s⊢l[::≼]L = G,s⊢l[::≼]L"*
by *(auto simp: lconf_def)*

lemma *lconf_new: "⟦L vn = None; G,s⊢l[::≼]L⟧ ⟹ G,s⊢l(vn↦v)[::≼]L"*
by *(auto simp: lconf_def)*

lemma *lconf_upd: "⟦G,s⊢l[::≼]L; G,s⊢v::≼T; L vn = Some T⟧ ⟹ G,s⊢l(vn↦v)[::≼]L"*
by *(auto simp: lconf_def)*

lemma *lconf_ext: "⟦G,s⊢l[::≼]L; G,s⊢v::≼T⟧ ⟹ G,s⊢l(vn↦v)[::≼]L(vn↦T)"*
by *(auto simp: lconf_def)*

lemma *lconf_map_sum [simp]:*
"G,s⊢l1 (+) l2[::≼]L1 (+) L2 = (G,s⊢l1[::≼]L1 ∧ G,s⊢l2[::≼]L2)"
apply *(unfold lconf_def)*
apply *safe*
apply *(case_tac [3] "n")*
apply *(force split add: sum.split)+*
done

lemma *lconf_ext_list [rule_format (no_asm)]: "⋀X. ⟦G,s⊢l[::≼]L⟧ ⟹*
∀vs Ts. nodups vns ⟶ length Ts = length vns ⟶ list_all2 (conf G s)
vs Ts ⟶ G,s⊢l(vns[↦]vs)[::≼]L(vns[↦]Ts)"
apply *(unfold lconf_def)*
apply *(induct_tac "vns")*
apply *clarsimp*
apply *clarsimp*
apply *(frule list_all2_lengthD)*
apply *clarsimp*
done

lemma *lconf_deallocL: "⟦G,s⊢l[::≼]L(vn↦T); L vn = None⟧ ⟹ G,s⊢l[::≼]L"*
apply *(simp only: lconf_def)*
apply *safe*
apply *(drule spec)*
apply *(drule ospec)*
apply *auto*
done

lemma *lconf_gext [elim]: "⟦G,s⊢l[::≼]L; s≤|s'⟧ ⟹ G,s'⊢l[::≼]L"*
apply *(simp only: lconf_def)*
apply *fast*
done

lemma *lconf_empty [simp, intro!]: "G,s⊢vs[::≼]empty"*
apply *(unfold lconf_def)*
apply *force*
done

lemma *lconf_init_vals [intro!]:*
"⋀X. ∀n. ∀T∈fs n:is_type G T ⟹ G,s⊢init_vals fs[::≼]fs"
apply *(unfold lconf_def)*
apply *force*
done

57 object conformance

constdefs

```
  oconf :: "prog ⇒ st ⇒ obj ⇒ oref ⇒ bool"  ("_,_⊢_::≼√_"  [71,71,71,71]
70)
          "G,s⊢obj::≼√r ≡ G,s⊢snd obj[::≼]var_tys G (fst obj) r ∧ (case
r of
                                Heap a ⇒ is_type G (obj_ty obj) | Stat
C ⇒ True)"

lemma oconf_def2:  "G,s⊢(oi,fs)::≼√r =
  (G,s⊢fs[::≼]var_tys G oi r ∧ (case r of Heap a ⇒ is_type G (obj_ty
(oi,fs)) | Stat C ⇒ True))"
by (simp add: oconf_def Let_def)

lemma oconf_is_type: "G,s⊢obj::≼√Heap a ⟹ is_type G (obj_ty obj)"
by (auto simp: oconf_def Let_def)

lemma oconf_lconf: "G2,s2⊢(oi2, fs2)::≼√r2 ⟹ G2,s2⊢fs2[::≼]var_tys
G2 oi2 r2"
apply (rule oconf_def2 [THEN iffD1 [THEN conjunct1]])
by assumption

lemma oconf_cong [simp]: "G,set_locals l s⊢obj::≼√r = G,s⊢obj::≼√r"
by (auto simp: oconf_def Let_def)


ML_setup {*
claset_ref()  := claset() delSWrapper "split_all_tac";
simpset_ref() := simpset() delloop "split_all_tac"
*}
lemma oconf_init_obj_lemma: "⟦⋀C c. class G C = Some c ⟹ unique (fields
G C);
  ⋀C c f m T. ⟦class G C = Some c; table_of (fields G C) f = Some (m,
T) ⟧
            ⟹ is_type G T;
 (case r of Heap a ⇒ is_type G (obj_ty (oi,fs)) | Stat C ⇒ is_class
G C)⟧ ⟹
  G,s⊢(oi, init_vals (var_tys G oi r))::≼√r"
apply (auto simp add: oconf_def2)
apply (drule_tac var_tys_Some_eq [THEN iffD1])
defer
apply (subst obj_ty_eq)
apply(auto dest!: fields_table_SomeD split add: sum.split_asm obj_tag.split_asm)
done
ML_setup {*
claset_ref()  := claset() addSbefore ("split_all_tac", split_all_tac);
simpset_ref() := simpset() addloop ("split_all_tac", split_all_tac)
*}
```

58 state conformance

constdefs

```
  conforms :: "state ⇒ env_ ⇒ bool"            (      "_::≼_"   [71,71]
70)
              "xs::≼E ≡ let (G, L) = E; s = snd xs; l = locals s in
          (∀r. ∀obj∈globs s r:           G,s⊢obj   ::≼√r) ∧
                                        G,s⊢l    [::≼]L   ∧
          (∀a. fst xs=Some(XcptLoc a) ⟶ G,s⊢Addr a::≼Class (SXcpt Throwable))"
```

59 conforms

lemma *conforms_globsD:*
"⟦(x, s)::≼(G, L); globs s r = Some (oi,fs)⟧ ⟹ G,s⊢(oi,fs)::≼√r"
by *(auto simp: conforms_def Let_def)*

lemma *conforms_localD: "(x, s)::≼(G, L) ⟹ G,s⊢locals s[::≼]L"*
by *(auto simp: conforms_def Let_def)*

lemma *conforms_XcptLocD: "⟦(x, s)::≼(G, L); x = Some (XcptLoc a)⟧ ⟹*

G,s⊢Addr a::≼Class (SXcpt Throwable)"
by *(auto simp: conforms_def Let_def)*

lemma *conforms_RefTD: "⟦G,s2⊢a'::≼RefT t; a' ≠ Null; (x2,s2) ::≼(G,*
L)⟧ ⟹
∃a oi vs. a' = Addr a ∧ globs s2 (Inl a) = Some (oi, vs) ∧
G⊢obj_ty (oi, vs)≼RefT t ∧ is_type G (obj_ty (oi, vs))"
apply *(drule_tac conf_RefTD)*
apply *clarsimp*
apply *(rule conforms_globsD [THEN oconf_is_type])*
apply *auto*
done

lemma *conforms_StdXcpt [iff]:*
"((Some (StdXcpt xn), s)::≼(G, L)) = (Norm s::≼(G, L))"
by *(auto simp: conforms_def)*

lemma *conforms_raise_if [iff]:*
"((raise_if c xn x, s)::≼(G, L)) = ((x, s)::≼(G, L))"
by *(auto simp: xcpt_if_def)*

lemma *conforms_NormI: "(x, s)::≼(G, L) ⟹ Norm s::≼(G, L)"*
by *(auto simp: conforms_def Let_def)*

lemma *conformsI: "⟦∀r. ∀(oi,fs)∈globs s r: G,s⊢(oi,fs)::≼√r;*

```
      G,s⊢locals s[::≼]L;
      ∀a. x = Some (XcptLoc a) ⟶ G,s⊢Addr a::≼Class (SXcpt Throwable)⟧
⟹
  (x, s)::≼(G, L)"
by (auto simp: conforms_def Let_def)

lemma conforms_xconf: "⟦(x, s)::≼(G,L);
 ∀a. x' = Some (XcptLoc a) ⟶ G,s⊢Addr a::≼Class (SXcpt Throwable)⟧
⟹
 (x',s)::≼(G,L)"
by (fast intro: conformsI elim: conforms_globsD conforms_localD)

lemma conforms_lupd: "⟦(x, s)::≼(G, L); L vn = Some T;
  G,s⊢v::≼T⟧ ⟹ (x, lupd(vn↦v)s)::≼(G, L)"
by (force intro: conformsI lconf_upd dest: conforms_globsD conforms_localD

                                          conforms_XcptLocD simp: oconf_def2)

lemmas conforms_allocL_aux = conforms_localD [THEN lconf_ext]

lemma conforms_allocL:
  "⟦(x, s)::≼(G, L); G,s⊢v::≼T⟧ ⟹ (x, lupd(vn↦v)s)::≼(G, L(vn↦T))"
by (force intro: conformsI dest: conforms_globsD
         elim: conforms_XcptLocD conforms_allocL_aux simp: oconf_def2)

lemmas conforms_deallocL_aux = conforms_localD [THEN lconf_deallocL]
lemma conforms_deallocL: "⋀s. ⟦s::≼(G, L(vn↦T)); L vn = None⟧ ⟹ s::≼(G,L)"
by (fast intro: conformsI dest: conforms_globsD
        elim: conforms_XcptLocD conforms_deallocL_aux)

lemma conforms_gext: "⟦(x, s)::≼(G,L); s≤|s';
  ∀r. ∀(oi,fs)∈globs s' r: G,s'⊢(oi,fs)::≼√r;
   locals s'=locals s⟧ ⟹ (x,s')::≼(G,L)"
by (force intro!: conformsI dest: conforms_localD conforms_XcptLocD)


lemma conforms_xgext:
  "⋀s s'. ⟦(x ,s)::≼(G,L); (x', s')::≼(G, L); s'≤|s⟧ ⟹ (x',s)::≼(G,L)"
apply (erule_tac conforms_xconf)
apply (fast dest: conforms_XcptLocD)
done

lemma conforms_gupd: "⋀obj. ⟦(x, s)::≼(G, L); G,s⊢obj::≼√r; s≤|gupd(r↦obj)s⟧
⟹
  (x, gupd(r↦obj)s)::≼(G, L)"
apply (rule conforms_gext)
apply    auto
apply (force dest: conforms_globsD simp add: oconf_def2)+
done
```

```
lemma conforms_upd_gobj: "⟦(x,s)::≼(G, L); globs s r = Some (oi, vs);

  var_tys G oi r n = Some T; G,s⊢v::≼T⟧ ⟹ (x,upd_gobj r n v s)::≼(G,L)"
apply (rule conforms_gext)
apply auto
apply (force dest: conforms_globsD intro!: lconf_upd
       simp add: oconf_def2 cong del: sum.weak_case_cong)+
done

lemma conforms_set_locals:
  "⟦(x,s)::≼(G, L'); G,s⊢l[::≼]L⟧ ⟹ (x,set_locals l s)::≼(G,L)"
apply (auto intro!: conformsI dest: conforms_globsD
            elim!: conforms_XcptLocD simp add: oconf_def2)
done

lemma conforms_return: "⋀s'. ⟦(x,s)::≼(G, L); (x',s')::≼(G, L'); s≤|s'⟧
⟹
  (x',set_locals (locals s) s')::≼(G, L)"
apply (rule conforms_xconf)
prefer 2 apply (force dest: conforms_XcptLocD)
apply (erule conforms_gext)
apply (force dest: conforms_globsD)+
done

end
```

```
theory TypeSafe = Eval + WellForm + Conform:
```

60 result conformance

```
constdefs
  assign_conforms :: "st ⇒ (val ⇒ state ⇒ state) ⇒ ty ⇒ env ⇒ bool"
         ("_≤|_≼_::≼_"                                          [71,71,71,71]
70)
 "s≤|f≼T::≼E ≡
  ∀s' w. Norm s'::≼E ⟶ fst E,s'⊢w::≼T ⟶ s≤|s' ⟶ assign f w (Norm
s')::≼E"

  rconf :: "prog ⇒ lenv ⇒ st ⇒ term ⇒ vals ⇒ tys ⇒ bool"
         ("_,_,_⊢_≻_::≼_"                                  [71,71,71,71,71,71]
70)
          "G,L,s⊢t≻v::≼T ≡ case T of
    Inl T  ⇒ if (∃vf. t=In2 vf)
               then G,s⊢fst (the_In2 v)::≼T ∧ s≤|snd (the_In2 v)≼T::≼(G,L)
               else G,s⊢the_In1 v::≼T
  | Inr Ts ⇒ list_all2 (conf G s) (the_In3 v) Ts"
```

lemma *rconf_In1 [simp]:*
"G,L,s⊢In1 ec≻In1 v ::≼Inl T = G,s⊢v::≼T"
apply *(unfold rconf_def)*
apply *(simp (no_asm))*
done
lemma *rconf_In2 [simp]:*
"G,L,s⊢In2 va≻In2 vf::≼Inl T = (G,s⊢fst vf::≼T ∧ s≤|snd vf≼T::≼(G,L))"
apply *(unfold rconf_def)*
apply *(simp (no_asm))*
done
lemma *rconf_In3 [simp]:*
"G,L,s⊢In3 es≻In3 vs::≼Inr Ts = list_all2 (λv T. G,s⊢v::≼T) vs Ts"
apply *(unfold rconf_def)*
apply *(simp (no_asm))*
done

61 fits conf

lemma *conf_fits: "G,s⊢v::≼T ⟹ G,s⊢v fits T"*
apply *(unfold fits_def)*
apply *clarify*
apply *(erule swap, simp (no_asm_use))*
apply *(drule conf_RefTD)*
apply *auto*
done

lemma *fits_conf: "⟦G,s⊢v::≼T; G⊢T≼? T'; G,s⊢v fits T'; ws_prog G⟧ ⟹ G,s⊢v::≼T'"*
apply *(auto dest!: fitsD cast_PrimT2 cast_RefT2)*
apply *(force dest: conf_RefTD intro: conf_AddrI)*
done

lemma *fits_Array:*
"⟦G,s⊢v::≼T; G⊢T'.[]≼T.[]; G,s⊢v fits T'; ws_prog G⟧ ⟹ G,s⊢v::≼T'"
apply *(auto dest!: fitsD widen_ArrayPrimT widen_ArrayRefT)*
apply *(force dest: conf_RefTD intro: conf_AddrI)*
done

62 gext

lemma *halloc_gext: "⋀s1 s2. G⊢s1 −halloc oi≻a→ s2 ⟹ snd s1≤|snd s2"*
apply *(simp (no_asm_simp) only: split_tupled_all)*
apply *(erule halloc.induct)*
apply *(auto dest!: new_AddrD)*
done

```
lemma sxalloc_gext: "⋀s1 s2. G⊢s1 −sxalloc→ s2 ⟹ snd s1≤|snd s2"
apply (simp (no_asm_simp) only: split_tupled_all)
apply (erule sxalloc.induct)
apply   (auto dest!: halloc_gext)
done

lemma eval_gext_lemma [rule_format (no_asm)]:
 "G⊢s −t≻→ (w,s') ⟹ snd s≤|snd s' ∧ (case w of
    In1 v ⇒ True
  | In2 vf ⇒ normal s ⟶ (∀v x s. s≤|snd (assign (snd vf) v (x,s)))

  | In3 vs ⇒ True)"
apply (erule eval_induct)
prefer 22
  apply (case_tac "inited C (globs s0)", clarsimp, erule thin_rl)
apply (auto del: conjI  dest!: not_initedD gext_new sxalloc_gext halloc_gext
 simp  add: lvar_def fvar_def2 avar_def2 init_lvars_def2
 split del: split_if_asm split add: sum3.split)

apply force+
done

lemma evar_gext_f: "G⊢Norm s1 −e=≻vf → s2 ⟹ s≤|snd (assign (snd
vf) v (x,s))"
apply (drule eval_gext_lemma [THEN conjunct2])
apply auto
done

lemmas eval_gext = eval_gext_lemma [THEN conjunct1]

lemma eval_gext': "G⊢(x1,s1) −t≻→ (w,x2,s2) ⟹ s1≤|s2"
apply (drule eval_gext)
apply auto
done

lemma init_yields_initd: "G⊢Norm s1 −init C→ s2 ⟹ initd C s2"
apply (erule eval_cases , auto split del: split_if_asm)
apply (case_tac "inited C (globs s1)")
apply  (clarsimp split del: split_if_asm)+
apply (drule eval_gext')+
apply (drule init_class_obj_inited)
apply (erule inited_gext)
apply (simp (no_asm_use))
done
```

63 Lemmas

```
lemma oconf_init_obj: "⟦wf_prog G;
 (case r of Heap a ⇒ is_type G (obj_ty (oi,fs)) | Stat C ⇒ is_class
```

```
G C)⟧ ⟹
  G,s⊢(oi, init_vals (var_tys G oi r))::≼√r"
apply (auto intro!: oconf_init_obj_lemma unique_fields)
done

lemma conforms_newG: "⟦globs s oref = None; (x, s)::≼(G,L);
  wf_prog G; case oref of Heap a ⇒ is_type G (obj_ty (oi,fs))
                        | Stat C ⇒ is_class G C⟧ ⟹
  (x, init_obj G oi oref s)::≼(G, L)"
apply (unfold init_obj_def)
apply (auto elim!: conforms_gupd oconf_init_obj)
done

lemma conforms_init_class_obj:
 "⟦(x,s)::≼(G, L); wf_prog G; class G C=Some y; ¬ inited C (globs s)⟧
⟹
  (x,init_class_obj G C s)::≼(G, L)"
apply (rule not_initedD [THEN conforms_newG])
apply    auto
done


lemma fst_init_lvars: "fst (init_lvars G C sig (invmode m e) a' pvs (x,s))
=
  (if m then x else (np a') x)"
apply (simp (no_asm) add: init_lvars_def2)
done
declare fst_init_lvars [simp]

lemma halloc_conforms: "⋀s1. ⟦G⊢s1 −halloc oi≻a→ s2; wf_prog G; s1::≼(G,
L);
  is_type G (obj_ty (oi,fs))⟧ ⟹ s2::≼(G, L)"
apply (simp (no_asm_simp) only: split_tupled_all)
apply (case_tac "aa")
apply  (auto elim!: halloc_elim_cases dest!: new_AddrD
       intro!: conforms_newG [THEN conforms_xconf] conf_AddrI)
done

lemma halloc_type_sound: "⋀s1. ⟦G⊢s1 −halloc oi≻a→ (x,s); wf_prog
G; s1::≼(G, L);
  T = obj_ty (oi,fs); is_type G T⟧ ⟹
  (x,s)::≼(G, L) ∧ (x = None ⟶ G,s⊢Addr a::≼T)"
apply (auto elim!: halloc_conforms)
apply (case_tac "aa")
apply (subst obj_ty_eq)
apply  (auto elim!: halloc_elim_cases dest!: new_AddrD intro!: conf_AddrI)
done

lemma sxalloc_type_sound:
```

```
 "⋀s1 s2. ⟦G⊢s1 −sxalloc→ s2; wf_prog G⟧ ⟹ case fst s1 of
  None ⇒ s2 = s1 | Some x ⇒
  (∃a. fst s2 = Some(XcptLoc a) ∧ (∀L. s1::≼(G,L) ⟶ s2::≼(G,L)))"
apply (simp (no_asm_simp) only: split_tupled_all)
apply (erule sxalloc.induct)
apply   auto
apply (rule halloc_conforms [THEN conforms_xconf])
apply     (auto elim!: halloc_elim_cases dest!: new_AddrD intro!: conf_AddrI)
done

lemma wt_init_comp_ty:
"is_type G T ⟹ (G,L)⊢init_comp_ty T::√"
apply (unfold init_comp_ty_def)
apply clarsimp
done

declare fun_upd_same [simp]

declare fun_upd_apply [simp del]
declare split_paired_All [simp del] split_paired_Ex [simp del]
declare split_if     [split del] split_if_asm     [split del]
        option.split [split del] option.split_asm [split del]
ML_setup {*
simpset_ref() := simpset() delloop "split_all_tac";
claset_ref () := claset () delSWrapper "split_all_tac"
*}

constdefs
  DynT_prop::"[prog,inv_mode,tname,ref_ty] ⇒ bool" ("_⊢_→_≼_"[71,71,71,71]70)
 "G⊢mode→D≼t ≡ mode = IntVir ⟶ is_class G D ∧
                    (if (∃T. t=ArrayT T) then D=Object else G⊢Class
D≼RefT t)"

lemma DynT_propI:
 "⟦(x,s)::≼(G, L); G,s⊢a'::≼RefT t; wf_prog G; mode = IntVir ⟶ a' ≠
Null⟧ ⟹
  G⊢mode→target mode s a' cT≼t"
apply (unfold DynT_prop_def)
apply clarsimp
apply (drule (2) conforms_RefTD)
apply clarsimp
apply (frule obj_ty_widenD)
apply (auto dest!: widen_Array widen_Array2 split add: split_if)
done

ML {*
fun exhaust_cmethd_tac s = EVERY'[
        res_inst_tac [("p",s)] PairE, rename_tac "md' j0",
        res_inst_tac [("p","j0")] PairE, rename_tac "j00 j1",
```

```
        res_inst_tac [("p","j1")] PairE, rename_tac "lvars body",
        res_inst_tac [("p","body")] PairE, rename_tac "blk res",
        res_inst_tac [("p","j00")] PairE, rename_tac "m' j2",
        res_inst_tac [("p","j2")] PairE, rename_tac "pns' rT'",
        hyp_subst_tac, K prune_params_tac]
*}
lemma DynT_mheadsD:
"⟦the (cmethd G dynT sig)=(md,(m',pns,rT),lvars,bdy); G⊢invmode m e→dynT≼t;

  wf_prog G; (G, L)⊢e::-RefT t; (cT, m, pnsa, T) ∈ mheads G t sig;
  target (invmode m e) s2 a' cT = dynT⟧ ⟹
  cmethd G dynT sig = Some (md, (m, pns, rT), lvars, bdy) ∧ G⊢rT≼T ∧
m' = m ∧
  wf_mdecl G md (sig, (m, pns, rT), lvars, bdy) ∧
  is_class G dynT ∧ G⊢dynT≼C md ∧ is_class G md ∧
 (invmode m e ≠ IntVir ⟶ (∃C. t=ClassT C ∧ G⊢C ≼C md) ∧ cT = ClassT
md)"
apply (unfold DynT_prop_def)
apply (frule (1) ty_expr_is_type)
apply (case_tac "invmode m e = IntVir")
apply  (clarsimp simp add: target_notIntVir)
defer
apply  (clarsimp simp add: target_notIntVir)
apply  (clarsimp simp add: invmode_IntVir_eq)
defer
apply  (clarsimp simp add: invmode_IntVir_eq)
apply  (drule (4) class_mheadsD)
defer
apply  (drule (3) static_mheadsD, clarsimp, frule cmethd_defpl,
        erule wf_ws_prog, simp (no_asm_simp))
apply  (clarsimp, drule (2) cmethd_wf_mdecl, clarsimp)+
done

lemma DynT_conf: "⟦G⊢target mode s2 a' cT≼C md; wf_prog G;
  G,s2⊢a'::≼RefT t; mode = IntVir ⟶ a' ≠ Null;
  mode ≠ IntVir ⟶ (∃C. t=ClassT C ∧ G⊢C≼C md) ∧ cT = ClassT md⟧⟹

  G,s2⊢a'::≼Class md"
apply (case_tac "mode ≠ IntVir")
apply  clarsimp
apply  (erule (1) widen.subcls [THEN conf_widen])
apply  (erule wf_ws_prog)
apply (drule conf_RefTD)
apply (auto intro!: conf_AddrI
            simp add: obj_class_def split add: obj_tag.split_asm)
done

lemma Ass_lemma: "⟦G⊢Norm s0 −va=≻(w, f)→ Norm s1; G⊢Norm s1 −e-≻v→
Norm s2; G,s2⊢v::≼T'; s1≤|s2 ⟶ assign f v (Norm s2)::≼(G, L)⟧ ⟹ assign
```

```
f v (Norm s2)::≼(G, L) ∧ (λ(x',s'). x' = None ⟶ G,s'⊢v::≼T') (assign
f v (Norm s2))"
apply (drule_tac x = "None" and s = "s2" and v = "v" in evar_gext_f)
apply (drule eval_gext', clarsimp)
apply (erule conf_gext)
apply simp
done

lemma Throw_lemma: "⟦G⊢tn≼C SXcpt Throwable; wf_prog G; (x1,s1)::≼(G,
L);
    x1 = None ⟶ G,s1⊢a'::≼Class tn⟧ ⟹ (throw a' x1, s1)::≼(G, L)"
apply (auto split add: split_xcpt_if simp add: throw_def2)
apply (erule conforms_xconf)
apply (frule conf_RefTD)
apply (auto elim: widen.subcls [THEN conf_widen])
done

lemma Try_lemma: "⟦G⊢obj_ty (the (globs s1' (Heap a)))≼ Class tn; (Some
(XcptLoc a), s1')::≼(G, L); wf_prog G⟧ ⟹ Norm (lupd(vn↦Addr a) s1')::≼(G,
L(vn↦Class tn))"
apply (rule conforms_allocL)
apply  (erule conforms_NormI)
apply (drule conforms_XcptLocD [THEN conf_RefTD],rule HOL.refl)
apply (auto intro!: conf_AddrI)
done

lemma Fin_lemma:
"⟦G⊢Norm s1 −c2→ (x2,s2); wf_prog G; (Some a, s1)::≼(G, L); (x2,s2)::≼(G,
L)⟧ ⟹
  (xcpt_if True (Some a) x2, s2)::≼(G, L)"
apply (force elim: eval_gext' conforms_xgext split add: split_xcpt_if)
done

lemma FVar_lemma1: "⟦table_of (fields G Ca) (fn, C) = Some (stat, fT);

  x2 = None ⟶ G,s2⊢a::≼Class Ca; wf_prog G; G⊢Ca≼C C; C ≠ Object;

  class G C = Some y; (x2,s2)::≼(G, L); s1≤|s2; inited C (globs s1);
  (if stat then id else np a) x2 = None⟧ ⟹
  ∃oi fs. globs s2 (if stat then Inr C else Inl (the_Addr a)) = Some (oi,fs)
∧
  var_tys G oi (if stat then Inr C else Inl(the_Addr a)) (Inl(fn,C)) =
Some fT"
apply (drule initedD)
apply (frule subcls_is_class2, simp (no_asm_simp))
apply (case_tac "stat")
apply  clarsimp
apply  (drule (1) rev_gext_objD, clarsimp)
apply  (frule fields_defpl, erule wf_ws_prog, simp (no_asm_simp))
```

```
apply  (rule var_tys_Some_eq [THEN iffD2])
apply  clarsimp
apply  (erule fields_table_SomeI, simp (no_asm))
apply clarsimp
apply (drule conf_RefTD, clarsimp, rule var_tys_Some_eq [THEN iffD2])
apply (auto dest!: widen_Array split add: obj_tag.split)
apply (rule fields_table_SomeI)
apply (auto elim!: fields_mono subcls_is_class2)
done

lemma FVar_lemma:
"⟦((v, f), Norm s2') = fvar C stat fn a (x2, s2); G⊢Ca≼C C;
  table_of (fields G Ca) (fn, C) = Some (stat, fT); wf_prog G;
  x2 = None ⟶ G,s2⊢a::≼Class Ca; C ≠ Object; class G C = Some y;
  (x2, s2)::≤(G, L); s1≤|s2; inited C (globs s1)⟧ ⟹
  G,s2'⊢v::≼fT ∧ s2'≤|f≼fT::≤(G, L)"
apply (unfold assign_conforms_def)
apply (drule sym)
apply (clarsimp simp add: fvar_def2)
apply (drule (9) FVar_lemma1)
apply (clarsimp, drule (2) conforms_globsD [THEN oconf_lconf, THEN lconfD])
apply clarsimp
apply (drule (1) rev_gext_objD,fast elim!: conforms_upd_gobj)
done

lemma AVar_lemma1: "⟦globs s2 (Inl a) = Some (Arr ty i, vs); the_Intg
i' in_bounds i; wf_prog G; G⊢ty.[]≼Tb.[]; Norm s2::≤(G, L)⟧ ⟹ G,s2⊢the
(vs (Inr (the_Intg i')))::≼Tb"
apply (erule widen_Array_Array [THEN conf_widen])
apply  (erule_tac [2] wf_ws_prog)
apply (drule (1) conforms_globsD [THEN oconf_lconf, THEN lconfD])
defer apply assumption
apply (force intro: var_tys_Some_eq [THEN iffD2])
done

lemma AVar_lemma: "⟦wf_prog G; G⊢(x1, s1) -e2-≻i→ (x2, s2);
  ((v,f), Norm s2') = avar G i a (x2, s2); x1 = None ⟶ G,s1⊢a::≼Ta.[];

  (x2, s2)::≤(G, L); s1≤|s2⟧ ⟹ G,s2'⊢v::≼Ta ∧ s2'≤|f≼Ta::≤(G, L)"
apply (unfold assign_conforms_def)
apply (drule sym)
apply (clarsimp simp add: avar_def2)
apply (drule (1) conf_gext)
apply (drule conf_RefTD, clarsimp)
apply (frule obj_ty_widenD)
apply (auto dest!: widen_Class)
apply  (erule (4) AVar_lemma1)
apply (auto split add: split_if)
apply (force elim!: fits_Array dest: gext_objD
```

```
      intro: var_tys_Some_eq [THEN iffD2] conforms_upd_gobj)
done
```

64 Call

```
lemma conforms_init_lvars_lemma: "⟦wf_prog G;
  wf_mhead G (mn,pTs) (m,pns,rT); Ball (set lvars) (split (λvn. is_type
G));
  list_all2 (conf G s2) pvs pTsa; G⊢pTsa[≼]pTs⟧ ⟹
  G,s2⊢init_vals (table_of lvars)(pns[↦]pvs)[::≼]table_of lvars(pns[↦]pTs)"
apply (unfold wf_mhead_def)
apply clarify
apply (erule (2) Ball_set_table [THEN lconf_init_vals, THEN lconf_ext_list])
apply (drule wf_ws_prog)
apply (erule (2) conf_list_widen)
done

lemma conforms_init_lvars: "⟦wf_mhead G (mn, pTs) (m, pns', rT'); wf_prog
G;
  Ball (set lvars) (split (λvn. is_type G));
  list_all2 (conf G s2) pvs pTsa; G⊢pTsa[≼] pTs;
  (x2, s2)::≼(G, L); cmethd G (target (invmode m e) s2 a' cT) (mn, pTs)
=
  Some (md, (ma, pns', rT'), lvars, blk, res);
  G⊢target (invmode m e) s2 a' cT≼C md; G,s2⊢a'::≼RefT t;
  invmode m e = IntVir ⟶ a' ≠ Null; invmode m e ≠ IntVir ⟶
  (∃C. t = ClassT C ∧ G⊢C≼C md) ∧ cT = ClassT md⟧ ⟹
  init_lvars G (target (invmode m e) s2 a' cT) (mn, pTs) (invmode m e)
a'
  pvs (x2,s2)::≼(G, table_of lvars(pns'[↦]pTs) (+) (if m then empty else
empty(()↦Class md)))"
apply (simp add: init_lvars_def2)
apply (rule conforms_set_locals)
apply  (simp (no_asm_simp) split add: split_if)
apply (drule (4) DynT_conf)
apply (case_tac "m")
apply  simp
defer  apply simp
apply  (rule conjI)
defer
apply   (rule lconf_ext, simp, assumption)
apply  (erule (4) conforms_init_lvars_lemma)+
done

lemma Call_type_sound: "⟦wf_prog G; G⊢(x1, s1) -ps≐≻pvs→ (x2, s2);

  C = target (invmode m e) s2 a' cT;
  s2' = init_lvars G C (mn, pTs) (invmode m e) a' pvs (x2,s2);
  G⊢s2' -Methd C (mn, pTs)-≻v→ (x3, s3);
```

```
  ∀L. s2'::≼(G, L) ⟶ (∀T. (G, L)⊢Methd C (mn, pTs)::-T ⟶
   (x3, s3)::≼(G, L) ∧ (x3 = None ⟶ G,s3⊢v::≼T));
  Norm s0::≼(G, L); (G, L)⊢e::-RefT t; (G, L)⊢ps::≐pTsa;
  max_spec G t (mn, pTsa) = {((cT, m, pns, rT), pTs)}; (x1, s1)::≼(G,
L);
  x1 = None ⟶ G,s1⊢a'::≼RefT t; (x2, s2)::≼(G, L);
  x2 = None ⟶ list_all2 (conf G s2) pvs pTsa⟧ ⟹
  (x3, set_locals (locals s2) s3)::≼(G, L) ∧ (x3 = None ⟶ G,s3⊢v::≼rT)"
apply clarify
apply (case_tac "x2")
defer
apply  (clarsimp split add: split_if_asm simp add: init_lvars_def2)
apply (case_tac "a' = Null ∧ ¬ m ∧ e ≠ Super")
apply  (clarsimp simp add: init_lvars_def2)
apply clarsimp
apply (drule eval_gext')
apply (frule (1) conf_gext)
apply (drule max_spec2mheads, clarsimp)
apply (subgoal_tac "invmode m e = IntVir ⟶ a' ≠ Null")
defer
apply  (clarsimp simp add: invmode_IntVir_eq)
apply (tactic {* exhaust_cmethd_tac "the (cmethd G (target (invmode m
e) s2 a' cT) (mn, pTs))" 1*})
apply (drule (7) DynT_mheadsD [OF _ DynT_propI], rule HOL.refl)
apply clarify
apply (drule wf_mdeclD1, clarsimp)
apply (drule (10) conforms_init_lvars, tactic "smp_tac 1 1")
apply (frule (2) wt_MethdI, clarsimp)
apply (tactic "smp_tac 1 1")
apply (clarify, rule conjI)
apply  (erule (1) conforms_return)
apply  (force dest!: eval_gext del: impCE simp add: init_lvars_def2)
apply clarsimp
apply (drule (2) widen_trans, erule (1) conf_widen)
apply (erule wf_ws_prog)
done
```

65 main proof of type safety

```
ML {*
val forward_hyp_tac = EVERY' [smp_tac 1,
        FIRST'[mp_tac,etac exI,smp_tac 1,EVERY'[etac impE,etac exI]],
        REPEAT o (etac conjE)];
val typD_tac = eresolve_tac (thms "wt_elim_cases") THEN_ALL_NEW
        EVERY' [full_simp_tac (simpset() setloop (K no_tac)),
         clarify_tac(claset() addSEs[])]
*}

lemma eval_type_sound [rule_format (no_asm)]:
```

```
 "wf_prog G ⟹ G⊢s0 −t≻→ (v,s1) ⟹ (∀L. s0::≼(G,L) ⟶
  (∀T. (G,L)⊢t::T ⟶ s1::≼(G,L) ∧
  (let (x,s) = s1 in x = None ⟶ G,L,s⊢t≻v::≼T)))"
apply (erule eval_induct)



apply          (simp_all (no_asm_use) add: Let_def body_def)
apply        (tactic "ALLGOALS (EVERY'[Clarify_tac, TRY o typD_tac,
                    TRY o forward_hyp_tac])")
apply       (tactic"ALLGOALS(EVERY'[asm_simp_tac(simpset()),TRY o Clarify_tac])")



apply (erule_tac V = "G⊢Norm s0 −?ea≻→ ?vs1" in thin_rl)
apply (frule eval_gext')
apply force


apply (force elim: conforms_localD [THEN lconfD] conforms_lupd
       simp add: assign_conforms_def lvar_def)


apply (force dest: fits_conf)


apply (rule conf_litval)
apply (simp add: empty_dt_def)


apply (rule conf_widen)
apply   (erule (1) subcls_direct [THEN widen.subcls])
apply  (erule (1) conforms_localD [THEN lconfD])
apply (erule wf_ws_prog)


apply fast




apply (simp split: split_if_asm)
apply  (tactic "forward_hyp_tac 1", force)
apply (tactic "forward_hyp_tac 1", force)


apply (force split add: split_if_asm)
```

```
apply (drule (1) wt.Loop)
apply (clarsimp split: split_if_asm)


apply (case_tac "x1", force)
apply (drule spec, erule impE, erule conforms_NormI)
apply (clarsimp)
apply (erule (3) Fin_lemma)


apply (erule (3) Throw_lemma)


apply (drule (2) halloc_type_sound, rule HOL.refl, simp, simp)


apply (tactic "smp_tac 1 1", frule wt_init_comp_ty, erule (1) notE impE)
apply (tactic "forward_hyp_tac 1")
apply (case_tac "check_neg i' ab")
apply  clarsimp
apply  (drule (2) halloc_type_sound, rule HOL.refl, simp, simp)
apply force




apply (case_tac "inited C (globs s0)")
apply  (clarsimp)
apply (clarsimp)
apply (frule (1) wf_prog_cdecl)
apply (drule spec, erule impE, erule (3) conforms_init_class_obj)
apply (drule_tac "psi" = "class G C = ?x" in asm_rl, erule impE,
 force dest!: wf_cdecl_supD split add: split_if)
apply (drule spec, erule impE, erule conforms_set_locals, rule lconf_empty)
apply (erule impE, erule wf_cdecl_wt_init)
apply (drule (1) conforms_return, force dest: eval_gext', assumption)



apply (tactic "forward_hyp_tac 1")
apply (rename_tac x1 s1 x2 s2 v va w L Ta T', case_tac x1)
prefer 2 apply force
apply (case_tac x2)
prefer 2 apply force
apply (simp, drule conjunct2)
apply (drule (1) conf_widen)
apply  (erule wf_ws_prog)
apply (erule (2) Ass_lemma)
apply (clarsimp simp add: assign_conforms_def)
```

```
apply (drule (1) sxalloc_type_sound, simp (no_asm_use))
apply (case_tac a)
apply  clarsimp
apply clarsimp
apply (tactic "smp_tac 1 1")
apply (simp split add: split_if_asm)
apply (fast dest: conforms_deallocL Try_lemma)


apply (frule cfield_fields)
apply (frule (1) ty_expr_is_type [THEN type_is_class])
apply (frule wf_ws_prog)
apply (frule (2) fields_defpl)
apply clarsimp

apply (tactic "smp_tac 1 1")
apply (tactic "forward_hyp_tac 1")
apply (rule conjI, force split add: split_if simp add: fvar_def2)
apply (drule init_yields_initd, frule eval_gext')
apply clarsimp
apply (case_tac "C=Object")
apply  clarsimp
apply (erule (9) FVar_lemma)


apply (tactic "forward_hyp_tac 1")
apply (erule_tac V = "G⊢Norm s0 -?e1-≻?a'→ (?x1 1, ?s1)" in thin_rl,

         frule eval_gext')
apply (rule conjI)
apply  (clarsimp simp add: avar_def2)
apply clarsimp
apply (erule (5) AVar_lemma)


apply (tactic "forward_hyp_tac 1")
apply (erule (1) Call_type_sound, (rule HOL.refl)+, assumption+)
done


declare fun_upd_apply [simp]
declare split_paired_All [simp] split_paired_Ex [simp]
declare split_if     [split] split_if_asm     [split]
        option.split [split] option.split_asm [split]
ML_setup {*
simpset_ref() := simpset() addloop ("split_all_tac", split_all_tac);
claset_ref()  := claset () addSbefore ("split_all_tac", split_all_tac)
```

```
*}

theorem eval_ts: "⟦G⊢s -e-≻v → (x',s'); wf_prog G; s::≼(G,L); (G,L)⊢e::-T⟧
⟹
  (x',s')::≼(G,L) ∧ (x'=None ⟶ G,s'⊢v::≼T)"
apply (drule (3) eval_type_sound)
apply (unfold Let_def)
apply clarsimp
done

theorem evals_ts: "⟦G⊢s -es≐≻vs→ (x',s'); wf_prog G; s::≼(G,L); (G,L)⊢es::≐Ts⟧
⟹
  (x',s')::≼(G,L) ∧ (x'=None ⟶ list_all2 (conf G s') vs Ts)"
apply (drule (3) eval_type_sound)
apply (unfold Let_def)
apply clarsimp
done

theorem evar_ts: "⟦G⊢s -v=≻vf→ (x',s'); wf_prog G; s::≼(G,L); (G,L)⊢v::=T⟧
⟹
  (x',s')::≼(G,L) ∧ (x'=None ⟶ G,L,s'⊢In2 v≻In2 vf::≼Inl T)"
apply (drule (3) eval_type_sound)
apply (unfold Let_def)
apply clarsimp
done

theorem exec_ts: "⟦G⊢s -s0→ s'; wf_prog G; s::≼(G,L); (G,L)⊢s0::√⟧
⟹ s'::≼(G,L)"
apply (drule (3) eval_type_sound)
apply (unfold Let_def)
apply clarsimp
done

theorem dyn_methods_understood:
 "⋀s. ⟦wf_prog G; (G,L)⊢{t,md,IntVir}e..mn({pTs'}ps)::-rT;
  s::≼(G,L); G⊢s -e-≻a'→ Norm s'; a' ≠ Null⟧ ⟹
  ∃a obj. a'=Addr a ∧ heap s' a = Some obj ∧ cmethd G (obj_class obj)
(mn, pTs') ≠ None"
apply (erule wt_elim_cases)
apply (drule max_spec2mheads)
apply (drule (3) eval_ts)
apply (clarsimp split del: split_if split_if_asm)
apply (drule (2) DynT_propI)
apply  (simp (no_asm_simp))
apply (tactic {* exhaust_cmethd_tac "the (cmethd G (target (invmode m
e) s' a' md) (mn, pTs'))" 1 *})
apply (drule (4) DynT_mheadsD [THEN conjunct1], rule HOL.refl)
apply (drule conf_RefTD)
apply clarsimp
```

done

end

theory *AxSem = Evaln + TypeSafe:*

types *res = vals*
syntax
Val :: "val ⇒ res"
Var :: "var ⇒ res"
Vals :: "val list ⇒ res"
translations
"Val x" => "(In1 x)"
"Var x" => "(In2 x)"
"Vals x" => "(In3 x)"

syntax
"Val_" :: "[pttrn] => pttrn" ("Val:_" [951] 950)
"Var_" :: "[pttrn] => pttrn" ("Var:_" [951] 950)
"Vals_" :: "[pttrn] => pttrn" ("Vals:_" [951] 950)
translations
"λVal:v . b" == "(λv. b) ∘ the_In1"
"λVar:v . b" == "(λv. b) ∘ the_In2"
"λVals:v. b" == "(λv. b) ∘ the_In3"

types *'a assn = "res ⇒ state ⇒ 'a ⇒ bool"*
translations
"res" <= (type) "AxSem.res"
"a assn" <= (type) "vals ⇒ state ⇒ a ⇒ bool"

constdefs
assn_imp :: "'a assn ⇒ 'a assn ⇒ bool" (**infixr** *"⇒" 25)*
"P ⇒ Q ≡ ∀Y s Z. P Y s Z ⟶ Q Y s Z"

lemma *assn_imp_def2 [iff]: "(P ⇒ Q) = (∀Y s Z. P Y s Z ⟶ Q Y s Z)"*
apply *(unfold assn_imp_def)*
apply *(rule HOL.refl)*
done

66 assertion transformers

66.1 peek$_and$

constdefs
```
  peek_and   :: "'a assn ⇒ (state ⇒  bool) ⇒ 'a assn" (infixl "∧." 13)
 "P ∧. p ≡ λY s Z. P Y s Z ∧ p s"
```

lemma *peek_and_def2 [simp]: "peek_and P p Y s = (λZ. (P Y s Z ∧ p s))"*
apply *(unfold peek_and_def)*
apply *(simp (no_asm))*
done

lemma *peek_and_Not [simp]: "(P ∧. (λs. ¬ f s)) = (P ∧. Not ∘ f)"*
apply *(rule ext)*
apply *(rule ext)*
apply *(simp (no_asm))*
done

lemma *peek_and_and [simp]: "peek_and (peek_and P p) p = peek_and P p"*
apply *(unfold peek_and_def)*
apply *(simp (no_asm))*
done

lemma *peek_and_commut: "(P ∧. p ∧. q) = (P ∧. q ∧. p)"*
apply *(rule ext)*
apply *(rule ext)*
apply *(rule ext)*
apply *auto*
done

syntax
```
  Normal     :: "'a assn ⇒ 'a assn"
```
translations
```
  "Normal P" == "P ∧. normal"
```

lemma *peek_and_Normal [simp]: "peek_and (Normal P) p = Normal (peek_and P p)"*
apply *(rule ext)*
apply *(rule ext)*
apply *(rule ext)*
apply *auto*
done

66.2 assn$_s$upd

constdefs
```
  assn_supd  :: "'a assn ⇒ (state ⇒ state) ⇒ 'a assn" (infixl ";." 13)
 "P ;. f ≡ λY s' Z. ∃s. P Y s Z ∧ s' = f s"
```

lemma *assn_supd_def2 [simp]: "assn_supd P f Y s' Z = (∃s. P Y s Z ∧ s' = f s)"*
apply *(unfold assn_supd_def)*
apply *(simp (no_asm))*
done

66.3 supd$_assn$

constdefs
supd_assn :: "(state ⇒ state) ⇒ 'a assn ⇒ 'a assn" (**infixr** *".;" 13)*
"f .; P ≡ λY s. P Y (f s)"

lemma *supd_assn_def2 [simp]: "(f .; P) Y s = P Y (f s)"*
apply *(unfold supd_assn_def)*
apply *(simp (no_asm))*
done

lemma *supd_assn_supdD [elim]: "((f .; Q) ;. f) Y s Z ⟹ Q Y s Z"*
apply *auto*
done

lemma *supd_assn_supdI [elim]: "Q Y s Z ⟹ (f .; (Q ;. f)) Y s Z"*
apply *(auto simp del: split_paired_Ex)*
done

66.4 subst$_res$

constdefs
subst_res :: "'a assn ⇒ res ⇒ 'a assn" *("_←_" [60,61] 60)*
"P←w ≡ λY. P w"

lemma *subst_res_def2 [simp]: "(P←w) Y = P w"*
apply *(unfold subst_res_def)*
apply *(simp (no_asm))*
done

lemma *subst_subst_res [simp]: "P←w←v = P←w"*
apply *(rule ext)*
apply *(simp (no_asm))*
done

lemma *peek_and_subst_res [simp]: "(P ∧. p)←w = (P←w ∧. p)"*
apply *(rule ext)*
apply *(rule ext)*
apply *(simp (no_asm))*
done

66.5 subst$_{B}ool$

constdefs
```
  subst_Bool  :: "'a assn ⇒ bool ⇒ 'a assn"             ("_ ←=_" [60,61]
60)
 "P←=b ≡ λY s Z. ∃v. P (Val v) s Z ∧ (normal s ⟶ the_Bool v=b)"
```

lemma *subst_Bool_def2 [simp]:*
```
"(P←=b) Y s Z = (∃v. P (Val v) s Z ∧ (normal s ⟶ the_Bool v=b))"
```
apply *(unfold subst_Bool_def)*
apply *(simp (no_asm))*
done

lemma *subst_Bool_the_BoolI: "P (Val b) s Z ⟹ (P←=the_Bool b) Y s
Z"*
apply *auto*
done

66.6 peek$_{r}es$

constdefs
```
  peek_res    :: "(res ⇒ 'a assn) ⇒ 'a assn"
 "peek_res Pf ≡ λY. Pf Y Y"
```

syntax
```
"@peek_res"  :: "pttrn ⇒ 'a assn ⇒ 'a assn"           ("λ_:. _" [0,3]
3)
```
translations
```
  "λw:. P"   == "peek_res (λw. P)"
```

lemma *peek_res_def2 [simp]: "peek_res P Y = P Y Y"*
apply *(unfold peek_res_def)*
apply *(simp (no_asm))*
done

lemma *peek_res_subst_res [simp]: "peek_res P←w = P w←w"*
apply *(rule ext)*
apply *(simp (no_asm))*
done

lemma *peek_subst_res_allI:*
```
 "(⋀a. T a (P (f a)←f a)) ⟹ ∀a. T a (peek_res P←f a)"
```
apply *(rule allI)*
apply *(simp (no_asm))*
apply *fast*
done

66.7 ign$_res$

constdefs
```
  ign_res    ::  "        'a assn ⇒ 'a assn"            ("_↓" [1000]
1000)
  "P↓        ≡ λY s Z. ∃Y. P Y s Z"
```

lemma *ign_res_def2 [simp]*: `"P↓ Y s Z = (∃Y. P Y s Z)"`
apply *(unfold ign_res_def)*
apply *(simp (no_asm))*
done

lemma *ign_ign_res [simp]*: `"P↓↓ = P↓"`
apply *(rule ext)*
apply *(rule ext)*
apply *(rule ext)*
apply *(simp (no_asm))*
done

lemma *ign_subst_res [simp]*: `"P↓←w = P↓"`
apply *(rule ext)*
apply *(rule ext)*
apply *(rule ext)*
apply *(simp (no_asm))*
done

lemma *peek_and_ign_res [simp]*: `"(P ∧. p)↓ = (P↓ ∧. p)"`
apply *(rule ext)*
apply *(rule ext)*
apply *(rule ext)*
apply *(simp (no_asm))*
done

66.8 peek$_s t$

constdefs
```
  peek_st    :: "(st ⇒ 'a assn) ⇒ 'a assn"
 "peek_st P ≡ λY s. P (snd s) Y s"
```

syntax
```
"@peek_st"   :: "pttrn ⇒ 'a assn ⇒ 'a assn"            ("λ_.. _" [0,3]
3)
```
translations
```
  "λs.. P"   == "peek_st (λs. P)"
```

lemma *peek_st_def2 [simp]*: `"(λs.. Pf s) Y s = Pf (snd s) Y s"`
apply *(unfold peek_st_def)*
apply *(simp (no_asm))*
done

lemma *peek_st_triv [simp]: "(λs.. P) = P"*
apply *(rule ext)*
apply *(rule ext)*
apply *(simp (no_asm))*
done

lemma *peek_st_st [simp]: "(λs.. λs'.. P s s') = (λs.. P s s)"*
apply *(rule ext)*
apply *(rule ext)*
apply *(simp (no_asm))*
done

lemma *peek_st_split [simp]: "(λs.. λY s'. P s Y s') = (λY s. P (snd s) Y s)"*
apply *(rule ext)*
apply *(rule ext)*
apply *(simp (no_asm))*
done

lemma *peek_st_subst_res [simp]: "(λs.. P s)←w = (λs.. P s←w)"*
apply *(rule ext)*
apply *(simp (no_asm))*
done

lemma *peek_st_Normal [simp]: "(λs..(Normal (P s))) = Normal (λs.. P s)"*
apply *(rule ext)*
apply *(rule ext)*
apply *(simp (no_asm))*
done

66.9 ign$_r es_e q$

constdefs
ign_res_eq :: "'a assn ⇒ res ⇒ 'a assn" *("_↓=_" [60,61] 60)*
"P↓=w ≡ λY:. P↓ ∧. (λs. Y=w)"

lemma *ign_res_eq_def2 [simp]: "(P↓=w) Y s Z = ((∃Y. P Y s Z) ∧ Y=w)"*
apply *(unfold ign_res_eq_def)*
apply *auto*
done

lemma *ign_ign_res_eq [simp]: "(P↓=w)↓ = P↓"*
apply *(rule ext)*
apply *(rule ext)*
apply *(rule ext)*
apply *(simp (no_asm))*
done

lemma *ign_res_eq_subst_res: "P↓=w←w = P↓"*
apply *(rule ext)*
apply *(rule ext)*
apply *(rule ext)*
apply *(simp (no_asm))*
done

lemma *subst_Bool_ign_res_eq: "((P←=b)↓=x) Y s Z = ((P←=b) Y s Z ∧ Y=x)"*
apply *(simp (no_asm))*
done

66.10 RefVar

constdefs
RefVar :: "(state ⇒ vvar × state) ⇒ 'a assn ⇒ 'a assn" (**infixr** *"..;" 13)*
"vf ..; P ≡ λY s. let (v,s') = vf s in P (Var v) s'"

lemma *RefVar_def2 [simp]: "(vf ..; P) Y s = P (Var (fst (vf s))) (snd (vf s))"*
apply *(unfold RefVar_def Let_def)*
apply *(simp (no_asm) add: split_beta)*
done

66.11 allocation

constdefs
Alloc :: "prog ⇒ obj_tag ⇒ 'a assn ⇒ 'a assn"
"Alloc G otag P ≡ λY s Z. ∀s' a. G⊢s −halloc otag≻a→ s'⟶ P (Val (Addr a)) s' Z"

SXAlloc :: "prog ⇒ 'a assn ⇒ 'a assn"
"SXAlloc G P ≡ λY s Z. ∀s'. G⊢s −sxalloc→ s' ⟶ P Y s' Z"

lemma *Alloc_def2 [simp]: "Alloc G otag P Y s Z = (∀s' a. G⊢s −halloc otag≻a→ s'⟶ P (Val (Addr a)) s' Z)"*
apply *(unfold Alloc_def)*
apply *(simp (no_asm))*
done

lemma *SXAlloc_def2 [simp]:*
"SXAlloc G P Y s Z = (∀s'. G⊢s −sxalloc→ s' ⟶ P Y s' Z)"
apply *(unfold SXAlloc_def)*
apply *(simp (no_asm))*

done

67 validity

constdefs

```
  type_ok  :: "prog ⇒ term ⇒ state ⇒ bool"
 "type_ok G t s ≡ ∃L T. (normal s ⟶ (G,L)⊢t::T) ∧ s::≼(G,L)"
```

datatype `'a triple = triple "('a assn)" "term" "('a assn)"`
`("{(1_)}/ _>/ {(1_)}" [3,65,3]75)`

types `'a triples = "'a triple set"`

syntax

```
  var_triple   :: "['a assn, var         ,'a assn] ⇒ 'a triple"
                                         ("{(1_)}/ _=>/ {(1_)}"    [3,80,3]
75)
  expr_triple  :: "['a assn, expr        ,'a assn] ⇒ 'a triple"
                                         ("{(1_)}/ _->/ {(1_)}"    [3,80,3]
75)
  exprs_triple :: "['a assn, expr list   ,'a assn] ⇒ 'a triple"
                                         ("{(1_)}/ _#>/ {(1_)}"    [3,65,3]
75)
  stmt_triple  :: "['a assn, stmt,        'a assn] ⇒ 'a triple"
                                         ("{(1_)}/ ._./ {(1_)}"     [3,65,3]
75)
```

syntax *(xsymbols)*

```
  triple       :: "['a assn, term        ,'a assn] ⇒ 'a triple"
                                         ("{(1_)}/ _≻/ {(1_)}"     [3,65,3]
75)
  var_triple   :: "['a assn, var         ,'a assn] ⇒ 'a triple"
                                         ("{(1_)}/ _=≻/ {(1_)}"    [3,80,3]
75)
  expr_triple  :: "['a assn, expr        ,'a assn] ⇒ 'a triple"
                                         ("{(1_)}/ _-≻/ {(1_)}"    [3,80,3]
75)
  exprs_triple :: "['a assn, expr list   ,'a assn] ⇒ 'a triple"
                                         ("{(1_)}/ _≐≻/ {(1_)}"    [3,65,3]
75)
```

translations

```
  "{P} e-≻ {Q}" == "{P} In1l e≻ {Q}"
  "{P} e=≻ {Q}" == "{P} In2  e≻ {Q}"
  "{P} e≐≻ {Q}" == "{P} In3  e≻ {Q}"
  "{P} .c. {Q}" == "{P} In1r c≻ {Q}"
```

lemma `inj_triple: "inj (λ(P,t,Q). {P} t≻ {Q})"`

apply *(rule injI)*
apply *auto*
done

lemma *triple_inj_eq: "({P} t≻ {Q} = {P'} t'≻ {Q'}) = (P=P' ∧ t=t' ∧ Q=Q')"*
apply *auto*
done

constdefs
```
  mtriples  :: "('c ⇒ 'sig ⇒ 'a assn) ⇒ ('c ⇒ 'sig ⇒ expr) ⇒
                ('c ⇒ 'sig ⇒ 'a assn) ⇒ ('c × 'sig) set ⇒ 'a triples"
                                        ("{{(1_)}/ _-≻/ {(1_)} | _}"[3,65,3,65]75)
 "{{P} tf-≻ {Q} | ms} ≡ (λ(C,sig). {Normal(P C sig)} tf C sig-≻ {Q C sig})‘ms"
```

consts

```
 triple_valid :: "prog ⇒ nat ⇒         'a triple  ⇒ bool"
                                                ("_⊨_:_" [61,0, 58] 57)
    ax_valids :: "prog ⇒ 'b triples ⇒ 'a triples ⇒ bool"
                                                ("_,_|⊨_"   [61,58,58] 57)
    ax_derivs :: "prog ⇒ ('b triples × 'a triples) set"
```

syntax

```
 triples_valid:: "prog ⇒ nat ⇒          'a triples ⇒ bool"
                                                ("_||=_:_" [61,0, 58] 57)
     ax_valid :: "prog ⇒  'b triples ⇒ 'a triple  ⇒ bool"
                                                ("_,_|=_"   [61,58,58] 57)
     ax_Derivs:: "prog ⇒  'b triples ⇒ 'a triples ⇒ bool"
                                                ("_,_||-_"   [61,58,58] 57)
     ax_Deriv :: "prog ⇒  'b triples ⇒ 'a triple  ⇒ bool"
                                                ("_,_|-_"   [61,58,58] 57)
```

syntax *(xsymbols)*

```
 triples_valid:: "prog ⇒ nat ⇒          'a triples ⇒ bool"
                                                ("_|⊨_:_" [61,0, 58] 57)
     ax_valid :: "prog ⇒  'b triples ⇒ 'a triple  ⇒ bool"
                                                ("_,_⊨_"   [61,58,58] 57)
```

```
    ax_Derivs:: "prog ⇒  'b triples ⇒ 'a triples ⇒ bool"
                                                ("_,_|⊢_"   [61,58,58]
57)
    ax_Deriv :: "prog ⇒  'b triples ⇒ 'a triple  ⇒ bool"
                                                ( "_,_⊢_"   [61,58,58]
57)

defs  triple_valid_def:  "G|⊨n:t  ≡ case t of {P} t≻ {Q} ⇒
                          ∀Y s Z. P Y s Z ⟶ type_ok G t s ⟶
                          (∀Y' s'. G⊢s −t≻−n→ (Y',s') ⟶ Q Y' s'
Z)"
translations          "G|⊨n:ts" == "Ball ts (triple_valid G n)"
defs   ax_valids_def:"G,A|⊨ts  ≡  ∀n. G|⊨n:A ⟶ G|⊨n:ts"
translations          "G,A |⊨t"  == "G,A|⊨{t}"
                      "G,A|⊢ts" == "(A,ts) ∈ ax_derivs G"
                      "G,A ⊢t"  == "G,A|⊢{t}"

lemma triple_valid_def2: "G|⊨n:{P} t≻ {Q} =
 (∀Y s Z. P Y s Z ⟶ (∃L. (normal s ⟶ (∃T. (G,L)⊢t::T)) ∧ s::≼(G,L))
⟶
  (∀Y' s'. G⊢s −t≻−n→ (Y',s') ⟶ Q Y' s' Z))"
apply (unfold triple_valid_def type_ok_def)
apply (simp (no_asm))
done


declare split_paired_All [simp del] split_paired_Ex [simp del]
declare split_if     [split del] split_if_asm     [split del]
        option.split [split del] option.split_asm [split del]
ML_setup {*
simpset_ref() := simpset() delloop "split_all_tac";
claset_ref () := claset () delSWrapper "split_all_tac"
*}


inductive "ax_derivs G" intros

  empty: " G,A|⊢{}"
  insert:"⟦G,A⊢t; G,A|⊢ts⟧ ⟹
          G,A|⊢insert t ts"

  asm:   "ts⊆A ⟹ G,A|⊢ts"


  weaken:"⟦G,A|⊢ts'; ts ⊆ ts'⟧ ⟹ G,A|⊢ts"

  conseq:"∀Y s Z . P  Y s Z  ⟶ (∃P' Q'. G,A⊢{P'} t≻ {Q'} ∧ (∀Y' s'.

         (∀Y   Z'. P' Y s Z' ⟶ Q' Y' s' Z') ⟶
```

```
                                   Q  Y' s' Z ))
                                          ⟹ G,A⊢{P } t≻ {Q }"

  hazard:"G,A⊢{P ∧. Not ∘ type_ok G t} t≻ {Q}"

  Xcpt:  "G,A⊢{P←(arbitrary3 t) ∧. Not ∘ normal} t≻ {P}"


  LVar:  " G,A⊢{Normal (λs.. P←Var (lvar vn s))} LVar vn=≻ {P}"

  FVar: "⟦G,A⊢{Normal P} .init C. {Q};
         G,A⊢{Q} e-≻ {λVal:a:. fvar C stat fn a ..; R}⟧ ⟹
                                   G,A⊢{Normal P} {C,stat}e..fn=≻ {R}"

  AVar:  "⟦G,A⊢{Normal P} e1-≻ {Q};
         ∀a. G,A⊢{Q←Val a} e2-≻ {λVal:i:. avar G i a ..; R}⟧ ⟹
                                   G,A⊢{Normal P} e1.[e2]=≻ {R}"


  NewC: "⟦G,A⊢{Normal P} .init C. {Alloc G (CInst C) Q}⟧ ⟹
                                   G,A⊢{Normal P} NewC C-≻ {Q}"

  NewA: "⟦G,A⊢{Normal P} .init_comp_ty T. {Q};  G,A⊢{Q} e-≻
         {λVal:i:. xupd (check_neg i) .; Alloc G (Arr T (the_Intg i))
R}⟧ ⟹
                                   G,A⊢{Normal P} New T[e]-≻ {R}"

  Cast: "⟦G,A⊢{Normal P} e-≻ {λVal:v:. λs..
         xupd (raise_if (¬G,s⊢v fits T) ClassCast) .; Q←Val v}⟧ ⟹
                                   G,A⊢{Normal P} Cast T e-≻ {Q}"

  Inst: "⟦G,A⊢{Normal P} e-≻ {λVal:v:. λs..
                 Q←Val (Bool (v≠Null ∧ G,s⊢v fits RefT T))}⟧ ⟹
                                   G,A⊢{Normal P} e InstOf T-≻ {Q}"

  Lit:                            "G,A⊢{Normal (P←Val v)} Lit v-≻ {P}"

  Super:" G,A⊢{Normal (λs.. P←Val (val_this s))} Super-≻ {P}"

  Acc:  "⟦G,A⊢{Normal P} va=≻ {λVar:(v,f):. Q←Val v}⟧ ⟹
                                   G,A⊢{Normal P} Acc va-≻ {Q}"

  Ass:  "⟦G,A⊢{Normal P} va=≻ {Q};
     ∀vf. G,A⊢{Q←Var vf} e-≻ {λVal:v:. assign (snd vf) v .; R}⟧ ⟹
                                   G,A⊢{Normal P} va:=e-≻ {R}"

  Cond: "⟦G,A ⊢{Normal P} e0-≻ {P'};
         ∀b. G,A⊢{P'←=b} (if b then e1 else e2)-≻ {Q}⟧ ⟹
                                   G,A⊢{Normal P} e0 ? e1 : e2-≻ {Q}"
```

```
  Call: "⟦G,A⊢{Normal P} e-≻ {Q}; ∀a. G,A⊢{Q←Val a} args≐≻ {R a};
         ∀a vs D l. G,A⊢{(R a←Vals vs ∧.
              (λs. D = target mode (snd s) a cT ∧ l = locals (snd s))
;.
              init_lvars G D (mn,pTs) mode a vs) ∧.
              (λs. normal s ⟶ G⊢mode→D≼t)}
             Methd D (mn,pTs)-≻ {set_lvars l .; S}⟧ ⟹
        G,A⊢{Normal P} {t,cT,mode}e..mn({pTs}args)-≻ {S}"

  Methd:"⟦G,A∪ {{P} Methd-≻ {Q} | ms} |⊢ {{P} body G-≻ {Q} | ms}⟧ ⟹
                                G,A|⊢{{P} Methd-≻  {Q} | ms}"

  Body: "⟦G,A⊢{Normal P} .init D. {Q}; G,A⊢{Q} .c. {R}; G,A⊢{R} e-≻
{S}⟧ ⟹
                                G,A⊢{Normal P} Body D c e-≻ {S}"


  Nil:                         "G,A⊢{Normal (P←Vals [])} []≐≻ {P}"

  Cons: "⟦G,A⊢{Normal P} e-≻ {Q};
         ∀v. G,A⊢{Q←Val v} es≐≻ {λVals:vs:. R←Vals (v#vs)}⟧ ⟹
                                G,A⊢{Normal P} e#es≐≻ {R}"


  Skip:                        "G,A⊢{Normal (P←•)} .Skip. {P}"

  Expr: "⟦G,A⊢{Normal P} e-≻ {Q←•}⟧ ⟹
                                G,A⊢{Normal P} .Expr e. {Q}"

  Comp: "⟦G,A⊢{Normal P} .c1. {Q};
         G,A⊢{Q} .c2. {R}⟧ ⟹
                                G,A⊢{Normal P} .c1;;c2. {R}"

  If:   "⟦G,A ⊢{Normal P} e-≻ {P'};
         ∀b. G,A⊢{P'←=b} .(if b then c1 else c2). {Q}⟧ ⟹
                                G,A⊢{Normal P} .If(e) c1 Else c2. {Q}"

  Loop: "⟦G,A⊢{P} e-≻ {P'}; G,A⊢{Normal (P'←=True)} .c. {P}⟧ ⟹
                            G,A⊢{P} .While(e) c. {(P'←=False)↓=•}"

  Throw:"⟦G,A⊢{Normal P} e-≻ {λVal:a:. xupd (throw a) .; Q←•}⟧ ⟹
                                G,A⊢{Normal P} .Throw e. {Q}"

  Try:  "⟦G,A⊢{Normal P} .c1. {SXAlloc G Q};
         G,A⊢{Q ∧. (λs.  G,s⊢catch C) ;. new_xcpt_var vn} .c2. {R};
             (Q ∧. (λs. ¬G,s⊢catch C)) ⇒ R⟧ ⟹
```

```
                                    G,A⊢{Normal P} .Try c1 Catch(C vn) c2.
{R}"

  Fin:  "⟦G,A⊢{Normal P} .c1. {Q};
      ∀x. G,A⊢{Q ∧. (λs. x = fst s) ;. xupd (λx. None)}
              .c2. {xupd (xcpt_if (x≠None) x) .; R}⟧ ⟹
                                 G,A⊢{Normal P} .c1 Finally c2. {R}"

  Done:                         "G,A⊢{Normal (P←• ∧. initd C)} .init C.
{P}"

  Init: "⟦the (class G C) = (sc,si,fs,ms,ini);
         G,A⊢{Normal ((P ∧. Not ∘ initd C) ;. supd (init_class_obj G
C))}
              .(if C = Object then Skip else init sc). {Q};
      ∀l. G,A⊢{Q ∧. (λs. l = locals (snd s)) ;. set_lvars empty}
              .ini. {set_lvars l .; R}⟧ ⟹
                                 G,A⊢{Normal (P ∧. Not ∘ initd C)} .init
C. {R}"

axioms
  polymorphic_conseq:
        "∀Y s Z . P  Y s Z  ⟶ (∃P' Q'. G,A⊢{P'} t≻ {Q'} ∧ (∀Y' s'.

        (∀Y   Z'. P' Y s Z' ⟶ Q' Y' s' Z') ⟶
                                 Q  Y' s' Z' ))
                                          ⟹ G,A⊢{P } t≻ {Q }"

  polymorphic_Loop:
        "⟦G,A⊢{P} e-≻ {P'}; G,A⊢{Normal (P'←=True)} .c. {P}⟧ ⟹
                               G,A⊢{P} .While(e) c. {(P'←=False)↓=•}"
```

68 rules derived by induction

```
lemma cut_valid: "⟦G,A'|⊨ts; G,A|⊨A'⟧ ⟹ G,A|⊨ts"
apply (unfold ax_valids_def)
apply fast
done


lemma ax_thin [rule_format (no_asm)]:
  "G,(A'::'a triple set)|⊢(ts::'a triple set) ⟹ ∀A. A' ⊆ A ⟶ G,A|⊢ts"
apply (erule ax_derivs.induct)
apply                (tactic "ALLGOALS(EVERY'[Clarify_tac,REPEAT o smp_tac
1])")
apply                (rule ax_derivs.empty)
apply               (erule (1) ax_derivs.insert)
apply              (fast intro: ax_derivs.asm)
```

```
apply            (fast intro: ax_derivs.weaken)
apply           (rule ax_derivs.conseq, intro strip, tactic "smp_tac 3
1",clarify, tactic "smp_tac 1 1",rule exI, rule exI, erule (1) conjI)

prefer 16
apply (rule ax_derivs.Methd, drule spec, erule mp, fast)
apply (tactic {* TRYALL (resolve_tac ((funpow 5 tl) (thms "ax_derivs.intros"))

                    THEN_ALL_NEW Blast_tac) *})
apply (erule ax_derivs.Call, clarify, blast, blast)
done

lemma ax_thin_insert: "G,(A::'a triple set)⊢(t::'a triple) ⟹ G,insert
x A⊢t"
apply (erule ax_thin)
apply fast
done

lemma subset_mtriples_iff:
  "ts ⊆ {{P} mb-≻ {Q} | ms} = (∃ms'. ms'⊆ms ∧  ts = {{P} mb-≻ {Q}
| ms'})"
apply (unfold mtriples_def)
apply (rule subset_image_iff)
done

lemma weaken:
 "G,(A::'a triple set)|⊢(ts'::'a triple set) ⟹ !ts. ts ⊆ ts' ⟶ G,A|⊢ts"
apply (erule ax_derivs.induct)

apply       (tactic "ALLGOALS strip_tac")
apply       (tactic {* ALLGOALS(REPEAT o (EVERY'[dtac (thm "subset_singletonD"),
         etac disjE, fast_tac (claset() addSIs [thm "ax_derivs.empty"])]))*})
apply       (tactic "TRYALL hyp_subst_tac")
apply       (simp, rule ax_derivs.empty)
apply      (drule subset_insertD)
apply      (blast intro: ax_derivs.insert)
apply     (fast intro: ax_derivs.asm)

apply   (fast intro: ax_derivs.weaken)
apply  (rule ax_derivs.conseq, clarify, tactic "smp_tac 3 1", blast)

apply (tactic {* TRYALL (resolve_tac ((funpow 5 tl) (thms "ax_derivs.intros"))

                  THEN_ALL_NEW Fast_tac) *})

apply (clarsimp simp add: subset_mtriples_iff)
apply (rule ax_derivs.Methd)
apply (drule spec)
apply (erule impE)
```

```
apply  (rule exI)
apply  (erule conjI)
apply  (rule HOL.refl)
oops
```

69 rules derived from conseq

```
lemma conseq12: "⟦G,A⊢{P'} t≻ {Q'};
 ∀Y s Z. P Y s Z ⟶ (∀Y' s'. (∀Y Z'. P' Y s Z' ⟶ Q' Y' s' Z') ⟶

  Q Y' s' Z)⟧
  ⟹  G,A⊢{P ::'a assn} t≻ {Q }"
apply (rule polymorphic_conseq)
apply clarsimp
apply blast
done


lemma conseq12': "⟦G,A⊢{P'} t≻ {Q'}; ∀s Y' s'.
       (∀Y Z. P' Y s Z ⟶ Q' Y' s' Z) ⟶
       (∀Y Z. P  Y s Z ⟶ Q  Y' s' Z)⟧
  ⟹  G,A⊢{P } t≻ {Q }"
apply (erule conseq12)
apply fast
done

lemma conseq12_from_conseq12': "⟦G,A⊢{P'} t≻ {Q'};
 ∀Y s Z. P Y s Z ⟶ (∀Y' s'. (∀Y Z'. P' Y s Z' ⟶ Q' Y' s' Z') ⟶

  Q Y' s' Z)⟧
  ⟹  G,A⊢{P } t≻ {Q }"
apply (erule conseq12')
apply blast
done

lemma conseq1: "⟦G,A⊢{P'} t≻ {Q}; P ⇒ P'⟧ ⟹ G,A⊢{P } t≻ {Q}"
apply (erule conseq12)
apply blast
done

lemma conseq2: "⟦G,A⊢{P} t≻ {Q'}; Q' ⇒ Q⟧ ⟹ G,A⊢{P} t≻ {Q}"
apply (erule conseq12)
apply blast
done

lemma ax_escape: "⟦∀Y s Z. P Y s Z ⟶ G,A⊢{λY' s' Z'. (Y',s') = (Y,s)}
t≻ {λY s Z'. Q Y s Z}⟧ ⟹
  G,A⊢{P} t≻ {Q}"
apply (rule polymorphic_conseq)
```

apply *force*
done

lemma *ax_constant:* "⟦ *C* ⟹ *G,A*⊢*{P} t*≻ *{Q}*⟧ ⟹ *G,A*⊢*{*λ*Y s Z. C* ∧ *P Y s Z} t*≻ *{Q}*"
apply *(rule ax_escape)*
apply *clarify*
apply *(rule conseq12)*
apply *fast*
apply *auto*
done

lemma *ax_impossible [intro]:* "*G,A*⊢*{*λ*Y s Z. False} t*≻ *{Q}*"
apply *(rule ax_escape)*
apply *clarify*
done

lemma *ax_nochange_lemma:* "⟦*P Y s; All (op = w)*⟧ ⟹ *P w s*"
apply *auto*
done
lemma *ax_nochange:*"*G,A*⊢*{*λ*Y s Z. (Y,s)=Z} t*≻ *{*λ*Y s Z. (Y,s)=Z}* ⟹ *G,A*⊢*{P} t*≻ *{P}*"
apply *(erule conseq12)*
apply *auto*
apply *(erule (1) ax_nochange_lemma)*
done

lemma *ax_trivial:* "*G,A*⊢*{P} t*≻ *{*λ*Y s Z. True}*"
apply *(rule polymorphic_conseq)*
apply *auto*
done

lemma *ax_disj:* "⟦*G,A*⊢*{P1} t*≻ *{Q1}; G,A*⊢*{P2} t*≻ *{Q2}*⟧ ⟹
G,A⊢*{*λ*Y s Z. P1 Y s Z* ∨ *P2 Y s Z} t*≻ *{*λ*Y s Z. Q1 Y s Z* ∨ *Q2 Y s Z}*"
apply *(rule ax_escape)*
apply *safe*
apply *(erule conseq12, fast)+*
done

lemma *ax_supd_shuffle:* "(∃*Q. G,A*⊢*{P} .c1. {Q}* ∧ *G,A*⊢*{Q ;. f} .c2. {R})*
=
(∃*Q'. G,A*⊢*{P} .c1. {f .; Q'}* ∧ *G,A*⊢*{Q'} .c2. {R})*"

```
apply (best elim!: conseq1 conseq2)
done

lemma ax_cases: "⟦G,A⊢{P ∧.       C} t≻ {Q};
                  G,A⊢{P ∧. Not ∘ C} t≻ {Q}⟧ ⟹ G,A⊢{P} t≻ {Q}"
apply (unfold peek_and_def)
apply (rule ax_escape)
apply clarify
apply (case_tac "C s")
apply  (erule conseq12, force)+
done


lemma peek_and_forget1_Normal:
 "G,A⊢{Normal P} t≻ {Q} ⟹ G,A⊢{Normal (P ∧. p)} t≻ {Q}"
apply (erule conseq1)
apply (simp (no_asm))
done

lemma peek_and_forget1: "G,A⊢{P} t≻ {Q} ⟹ G,A⊢{P ∧. p} t≻ {Q}"
apply (erule conseq1)
apply (simp (no_asm))
done

lemmas ax_NormalD = peek_and_forget1 [of _ _ _ _ _ normal]

lemma peek_and_forget2: "G,A⊢{P} t≻ {Q ∧. p} ⟹ G,A⊢{P} t≻ {Q}"
apply (erule conseq2)
apply (simp (no_asm))
done

lemma ax_subst_Val_allI: "∀v. G,A⊢{(P'              v )←Val v} t≻
{Q v} ⟹
     ∀v. G,A⊢{(λw:. P' (the_In1 w))←Val v} t≻ {Q v}"
apply (force elim!: conseq1)
done

lemma ax_subst_Var_allI: "∀v. G,A⊢{(P'              v )←Var v} t≻
{Q v} ⟹
     ∀v. G,A⊢{(λw:. P' (the_In2 w))←Var v} t≻ {Q v}"
apply (force elim!: conseq1)
done

lemma ax_subst_Vals_allI: "(∀v. G,A⊢{(     P'          v )←Vals v}
t≻ {Q v}) ⟹
      ∀v. G,A⊢{(λw:. P' (the_In3 w))←Vals v} t≻ {Q v}"
apply (force elim!: conseq1)
done
```

70 adaptation completeness

constdefs
```
  adapt_pre :: "'a assn ⇒ 'a assn ⇒ 'a assn ⇒ 'a assn"
 "adapt_pre P Q Q' ≡ λY s Z. ∀Y' s'. ∃Z'. P Y s Z' ∧ (Q Y' s' Z' ⟶
Q' Y' s' Z)"
```

lemma *ax_adapt: "G,A⊢{P} t≻ {Q} ⟹ G,A⊢{adapt_pre P Q Q'} t≻ {Q'}"*
apply *(unfold adapt_pre_def)*
apply *(erule conseq12)*
apply *fast*
done

lemma *adapt_pre_adapts: "G,A⊨{P} t≻ {Q} ⟶ G,A⊨{adapt_pre P Q Q'}*
t≻ {Q'}"
apply *(unfold adapt_pre_def)*
apply *(simp add: ax_valids_def triple_valid_def2)*
apply *fast*
done

lemma *adapt_pre_weakest:*
"∀G (A::'a triple set) t. G,A⊨{P} t≻ {Q} ⟶ G,A⊨{P'} t≻ {Q'} ⟹

* P' ⇒ adapt_pre P Q (Q'::'a assn)"*
apply *(unfold adapt_pre_def)*
apply *(drule spec)*
apply *(drule_tac x = "{}"* **in** *spec)*
apply *(drule_tac x = "In1r Skip"* **in** *spec)*
apply *(simp add: ax_valids_def triple_valid_def2)*
oops

71 alternative axioms

lemma *ax_Lit2:*
* "G,(A::'a triple set)⊢{Normal P::'a assn} Lit v-≻ {Normal (P↓=Val v)}"*
apply *(rule ax_derivs.Lit [THEN conseq1])*
apply *force*
done
lemma *ax_Lit2_test_complete:*
* "G,(A::'a triple set)⊢{Normal (P←Val v)::'a assn} Lit v-≻ {P}"*
apply *(rule ax_Lit2 [THEN conseq2])*
apply *force*
done

lemma *ax_LVar2: "G,(A::'a triple set)⊢{Normal P::'a assn} LVar vn=≻*
{Normal (λs.. P↓=Var (lvar vn s))}"
apply *(rule ax_derivs.LVar [THEN conseq1])*
apply *force*

done

lemma *ax_Super2: "G,(A::'a triple set)⊢*
{Normal P::'a assn} Super-≻ {Normal (λs.. P↓=Val (val_this s))}"
apply *(rule ax_derivs.Super [THEN conseq1])*
apply *force*
done

lemma *ax_Nil2:*
"G,(A::'a triple set)⊢{Normal P::'a assn} []≐≻ {Normal (P↓=Vals [])}"
apply *(rule ax_derivs.Nil [THEN conseq1])*
apply *force*
done

72 misc derived structural rules

lemma *ax_finite_mtriples_lemma: "⟦F ⊆ ms; finite ms; ∀(C,sig)∈ms.*
G,(A::'a triple set)⊢{Normal (P C sig)::'a assn} mb C sig-≻ {Q C sig}⟧ ⟹
G,A|⊢{{P} mb-≻ {Q} | F}"
apply *(frule (1) finite_subset)*
apply *(erule make_imp)*
apply *(erule thin_rl)*
apply *(erule finite_induct)*
apply *(unfold mtriples_def)*
apply *(clarsimp intro!: ax_derivs.empty ax_derivs.insert)+*
apply *force*
done
lemmas *ax_finite_mtriples = ax_finite_mtriples_lemma [OF subset_refl]*

lemma *ax_derivs_insertD:*
"G,(A::'a triple set)|⊢insert (t::'a triple) ts ⟹ G,A⊢t ∧ G,A|⊢ts"
apply *(fast intro: ax_derivs.weaken)*
done

lemma *ax_methods_spec:*
"⟦G,(A::'a triple set)|⊢split f ‘ ms; (C,sig) ∈ ms⟧⟹ G,A⊢((f C sig)::'a triple)"
apply *(erule ax_derivs.weaken)*
apply *(force del: image_eqI intro: rev_image_eqI)*
done

lemma *ax_finite_pointwise_lemma [rule_format]: "⟦F ⊆ ms; finite ms⟧*
⟹
((∀(C,sig)∈F. G,(A::'a triple set)⊢(f C sig::'a triple)) ⟶ (∀(C,sig)∈ms. G,A⊢(g C sig::'a triple))) ⟶
G,A|⊢split f ‘ F ⟶ G,A|⊢split g ‘ F"
apply *(frule (1) finite_subset)*

```
apply (erule make_imp)
apply (erule thin_rl)
apply (erule finite_induct)
apply  clarsimp+
apply (drule ax_derivs_insertD)
apply (rule ax_derivs.insert)
apply  (simp (no_asm_simp) only: split_tupled_all)
apply  (auto elim: ax_methods_spec)
done
lemmas ax_finite_pointwise = ax_finite_pointwise_lemma [OF subset_refl]

lemma ax_no_hazard:
  "G,(A::'a triple set)⊢{P ∧. type_ok G t} t≻ {Q::'a assn} ⟹ G,A⊢{P}
t≻ {Q}"
apply (erule ax_cases)
apply (rule ax_derivs.hazard [THEN conseq1])
apply force
done

lemma ax_free_wt:
 "(∃T L. (G,L)⊢t::T) ⟶ G,(A::'a triple set)⊢{Normal P} t≻ {Q::'a assn}
⟹
  G,A⊢{Normal P} t≻ {Q}"
apply (rule ax_no_hazard)
apply (rule ax_escape)
apply clarify
apply (erule mp [THEN conseq12])
apply  (auto simp add: type_ok_def)
done

ML {*
bind_thms ("ax_Xcpts", sum3_instantiate (thm "ax_derivs.Xcpt"))
*}
declare ax_Xcpts [intro!]

lemmas ax_Normal_cases = ax_cases [of _ _ normal]

lemma ax_Skip [intro!]: "G,(A::'a triple set)⊢{P←•} .Skip. {P::'a assn}"
apply (rule ax_Normal_cases)
apply  (rule ax_derivs.Skip)
apply fast
done
lemmas ax_SkipI = ax_Skip [THEN conseq1, standard]
```

73 derived rules for methd call

```
lemma ax_Call_Static:
 "⟦∀vs l. G,A⊢{R←Vals vs ∧. (λs. l = locals (snd s)) ;.
              init_lvars G C (mn,pTs) Static any_Addr vs}
```

```
            Methd C (mn,pTs)-≻ {set_lvars l .; S};
       G,A⊢{Normal P} e-≻ {Q}; G,(A::'a triple set)⊢{Q↓} args≐≻ {R::'a
assn}⟧ ⟹
       G,A⊢{Normal P} {t,ClassT C,Static}e..mn({pTs}args)-≻ {S}"
apply (erule ax_derivs.Call)
apply  safe
apply  (erule conseq1)
apply  force
apply (rule ax_escape, clarsimp)
apply (erule_tac V = "?P ⟶ ?Q" in thin_rl)
apply (drule spec, drule spec, erule conseq12)
apply (force simp add: init_lvars_def)
done

lemma ax_Call_known_DynT:
"⟦G⊢IntVir→C≼t; ∀a vs l. G,A⊢{(R a←Vals vs ∧. (λs. l = locals (snd
s)) ;.
  init_lvars G C (mn,pTs) IntVir a vs)} Methd C (mn,pTs)-≻ {set_lvars
l .; S};
  ∀a. G,A⊢{Q←Val a} args≐≻
          {R a ∧. (λs. C = obj_class (the (heap (snd s) (the_Addr a))))};

       G,(A::'a triple set)⊢{Normal P} e-≻ {Q::'a assn}⟧
   ⟹ G,A⊢{Normal P} {t,cT,IntVir}e..mn({pTs}args)-≻ {S}"
apply (erule ax_derivs.Call)
apply  safe
apply  (erule spec)
apply (rule ax_escape, clarsimp)
apply (drule spec, drule spec, drule spec, erule conseq12)
apply force
done

lemma ax_Methd1:
 "⟦G,A∪{{P} Methd-≻ {Q} | ms}|⊢ {{P} body G-≻ {Q} | ms}; (C,sig)∈ ms⟧
⟹
       G,A⊢{Normal (P C sig)} Methd C sig-≻ {Q C sig}"
apply (drule ax_derivs.Methd)
apply (unfold mtriples_def)
apply (erule (1) ax_methods_spec)
done

lemma ax_MethdN:
"G,insert({Normal P} Methd  C sig-≻ {Q}) A⊢
          {Normal P} body G C sig-≻ {Q} ⟹
      G,A⊢{Normal P} Methd   C sig-≻ {Q}"
apply (rule ax_Methd1)
apply  (rule_tac [2] singletonI)
apply (unfold mtriples_def)
apply clarsimp
```

done

lemma *ax_StatRef:*
"G,(A::'a triple set)⊢{Normal (P←Val Null)} StatRef rt-≻ {P::'a assn}"
apply *(rule ax_derivs.Cast)*
apply *(rule ax_Lit2 [THEN conseq2])*
apply *clarsimp*
done

74 rules derived from Init and Done

lemma *ax_InitS: "⟦the (class G C) = (sc,si,fs,ms,ini); C ≠ Object;*

∀l. G,A⊢{Q ∧. (λs. l = locals (snd s)) ;. set_lvars empty}
.ini. {set_lvars l .; R};
G,A⊢{Normal ((P ∧. Not ∘ initd C) ;. supd (init_class_obj G C))}
.init sc. {Q}⟧ ⟹
G,(A::'a triple set)⊢{Normal (P ∧. Not ∘ initd C)} .init C. {R::'a assn}"
apply *(erule ax_derivs.Init)*
apply *(simp (no_asm_simp))*
apply *assumption*
done

lemma *ax_Init_Skip_lemma:*
"∀l. G,(A::'a triple set)⊢{P←• ∧. (λs. l = locals (snd s)) ;. set_lvars l'}
.Skip. {(set_lvars l .; P)::'a assn}"
apply *(rule allI)*
apply *(rule ax_SkipI)*
apply *clarsimp*
done

lemma *ax_triv_InitS: "⟦the (class G C) = (sc,si,fs,ms,Skip); C ≠ Object;*

P←• ⇒ (supd (init_class_obj G C) .; P);
G,A⊢{Normal (P ∧. initd C)} .init sc. {(P ∧. initd C)←•}⟧ ⟹

G,(A::'a triple set)⊢{Normal P←•} .init C. {(P ∧. initd C)::'a assn}"
apply *(rule_tac C = "initd C"* **in** *ax_cases)*
apply *(rule conseq1, rule ax_derivs.Done, clarsimp)*
apply *(simp (no_asm))*
apply *(erule (1) ax_InitS)*
apply *(rule ax_Init_Skip_lemma)*
apply *(erule conseq1)*
apply *force*
done

```
lemma ax_Init_Object: "wf_prog G ⟹ G,(A::'a triple set)⊢
  {Normal ((supd (init_class_obj G Object) .; P←•) ∧. Not ∘ initd Object)}

       .init Object. {(P ∧. initd Object)::'a assn}"
apply (rule ax_derivs.Init)
apply   (drule class_Object, force)
apply  (rule_tac [2] ax_Init_Skip_lemma)
apply (simp (no_asm))
apply (rule ax_SkipI, clarsimp)
done

lemma ax_triv_Init_Object: "⟦wf_prog G;
       (P::'a assn) ⇒ (supd (init_class_obj G Object) .; P)⟧ ⟹
  G,(A::'a triple set)⊢{Normal P←•} .init Object. {P ∧. initd Object}"
apply (rule_tac C = "initd Object" in ax_cases)
apply  (rule conseq1, rule ax_derivs.Done, clarsimp)
apply (erule ax_Init_Object [THEN conseq1])
apply force
done
```

75 introduction rules for Alloc and SXAlloc

```
lemma ax_SXAlloc_Normal: "G,A⊢{P} .c. {Normal Q} ⟹ G,A⊢{P} .c. {SXAlloc
G Q}"
apply (erule conseq2)
apply (clarsimp elim!: sxalloc_elim_cases simp add: split_tupled_all)
done

lemma ax_Alloc:
  "G,A⊢{P} t≻ {Normal (λY (x,s) Z. (∀a. new_Addr (heap s) = Some a ⟶

 Q (Val (Addr a)) (Norm(init_obj G (CInst C) (Heap a) s)) Z)) ∧. heap_free
2}
  ⟹ G,A⊢{P} t≻ {Alloc G (CInst C) Q}"
apply (erule conseq2)
apply (auto elim!: halloc_elim_cases)
done

lemma ax_Alloc_Arr: "G,A⊢{P} t≻ {λVal:i:. Normal (λY (x,s) Z. ¬the_Intg
i<#0 ∧
  (∀a. new_Addr (heap s) = Some a ⟶
  Q (Val (Addr a)) (Norm (init_obj G (Arr T (the_Intg i)) (Heap a) s))
Z)) ∧.
   heap_free 2} ⟹
 G,A⊢{P} t≻ {λVal:i:. xupd (check_neg i) .; Alloc G (Arr T(the_Intg i))
Q}"
apply (erule conseq2)
apply (auto elim!: halloc_elim_cases)
```

done

lemma *ax_SXAlloc_catch_SXcpt: "⟦G,A⊢{P} t≻ {(λY (x,s) Z. x=Some (StdXcpt xn) ∧*
(∀a. new_Addr (heap s) = Some a ⟶
Q Y (Some (XcptLoc a),init_obj G (CInst (SXcpt xn)) (Heap a) s) Z))

∧. heap_free 2}⟧ ⟹
G,A⊢{P} t≻ {SXAlloc G (λY s Z. Q Y s Z ∧ G,s⊢catch SXcpt xn)}"
apply *(erule conseq2)*
apply *(auto elim!: sxalloc_elim_cases halloc_elim_cases)*
done

end

theory *AxExample = AxSem + Example:*

constdefs
arr_inv :: "st ⇒ bool"
"arr_inv ≡ λs. ∃obj a T el. globs s (Stat Base) = Some obj ∧
snd obj (Inl (arr, Base)) = Some (Addr a)
∧
heap s a = Some (Arr T #2,el)"

lemma *arr_inv_new_obj:*
"⋀a. ⟦arr_inv s; new_Addr (heap s)=Some a⟧ ⟹ arr_inv (gupd(Inl a↦x) s)"
apply *(unfold arr_inv_def)*
apply *(force dest!: new_AddrD2)*
done

lemma *arr_inv_set_locals [simp]: "arr_inv (set_locals l s) = arr_inv s"*
apply *(unfold arr_inv_def)*
apply *(simp (no_asm))*
done

lemma *arr_inv_gupd_Stat [simp]:*
"Base ≠ C ⟹ arr_inv (gupd(Stat C↦obj) s) = arr_inv s"
apply *(unfold arr_inv_def)*
apply *(simp (no_asm_simp))*
done

lemma *ax_inv_lupd [simp]: "arr_inv (lupd(x↦y) s) = arr_inv s"*
apply *(unfold arr_inv_def)*
apply *(simp (no_asm))*
done

```
declare split_if_asm [split del]
declare lvar_def [simp]

ML {*
fun inst1_tac s t = instantiate_tac [(s,t)];
val ax_tac = REPEAT o rtac allI THEN'
             resolve_tac(thm "ax_Skip"::thm "ax_StatRef"::thm "ax_MethdN"::
                         thm "ax_Alloc"::thm "ax_Alloc_Arr"::
                         thm "ax_SXAlloc_Normal"::
                         funpow 7 tl (thms "ax_derivs.intros"))
*}

theorem ax_test: "tprg,({}::'a triple set)⊢
  {Normal (λY s Z::'a. heap_free 4 s ∧ ¬initd Base s ∧ ¬ initd Ext s)}

  .test [Class Base]. {λY s Z. fst s = Some (StdXcpt IndOutBound)}"
apply (unfold test_def arr_viewed_from_def)
apply (tactic "ax_tac 1" )
defer
apply  (tactic "ax_tac 1" )
defer
apply    (tactic {* inst1_tac "Q1"
                 "λY s Z. arr_inv (snd s) ∧ tprg,s⊢catch SXcpt NullPointer"
*})
prefer 2
apply    simp
apply   (rule_tac P' = "Normal (λY s Z. arr_inv (snd s))" in conseq1)
prefer 2
apply    clarsimp
apply   (rule_tac Q' = "(λY s Z. ?Q Y s Z)←=False↓=•" in conseq2)
prefer 2
apply    simp
apply   (tactic "ax_tac 1" )
prefer 2
apply    (rule ax_impossible [THEN conseq1], clarsimp)
apply   (rule_tac P' = "Normal ?P" in conseq1)
prefer 2
apply    clarsimp
apply   (tactic "ax_tac 1")
apply   (tactic "ax_tac 1" )
prefer 2
apply    (rule ax_subst_Val_allI)
apply    (tactic {* inst1_tac "P'21" "λu a. Normal (?PP a←?x) u" *})
apply    (simp del: avar_def2 peek_and_def2)
apply    (tactic "ax_tac 1")
apply   (tactic "ax_tac 1")

apply   (rule_tac Q' = "Normal (λVar:(v, f) u ua. fst (snd (avar tprg
```

```
(Intg #2) v u)) = Some (StdXcpt IndOutBound))" in conseq2)
prefer 2
apply    (clarsimp simp add: split_beta)
apply   (tactic "ax_tac 1" )
apply    (tactic "ax_tac 2" )
apply   (rule ax_derivs.Done [THEN conseq1])
apply   (clarsimp simp add: arr_inv_def inited_def in_bounds_def)
defer
apply  (rule ax_SXAlloc_catch_SXcpt)
apply  (rule_tac Q' = "(λY (x, s) Z. x = Some (StdXcpt NullPointer) ∧
arr_inv s) ∧. heap_free 2" in conseq2)
prefer 2
apply   (simp add: arr_inv_new_obj)
apply  (tactic "ax_tac 1")
apply  (rule_tac C = "Ext" in ax_Call_known_DynT)
apply     (unfold DynT_prop_def)
apply     (simp (no_asm))
apply    (intro strip)
apply    (rule_tac P' = "Normal ?P" in conseq1)
apply     (tactic "ax_tac 1" )
apply     (rule ax_thin [OF _ empty_subsetI])
apply     (simp (no_asm) add: body_def2)
apply     (tactic "ax_tac 1" )
apply       (rule_tac [3] ax_derivs.Xcpt)
defer
apply      (simp (no_asm))
apply      (tactic "ax_tac 1")
apply      (tactic "ax_tac 1" )
prefer 2
apply       (rule ax_subst_Var_allI)
apply       (tactic {* inst1_tac "P'29" "λa vs l vf. ?PP a vs l vf←?x
∧. ?p" *})
apply       (rule allI)
apply       (tactic {* simp_tac (simpset() delloop "split_all_tac" delsimps
[thm "peek_and_def2"]) 1 *})
apply       (rule ax_derivs.Xcpt)
apply      (simp (no_asm))
apply      (tactic "ax_tac 1" )
apply       (tactic "ax_tac 2", tactic "ax_tac 2", tactic "ax_tac 2")
apply      (tactic "ax_tac 1")
apply     clarsimp
apply     (tactic {* inst1_tac "R16" "λa'. Normal ((λVals:vs (x, s) Z.
arr_inv s ∧ inited Ext (globs s) ∧ a' ≠ Null ∧ hd vs = Null) ∧. heap_free
2)" *})
prefer 5
apply     (rule ax_derivs.Done [THEN conseq1], force)
apply    force
apply   (rule ax_subst_Val_allI)
apply   (tactic {* inst1_tac "P'36" "λu a. Normal (?PP a←?x) u" *})
```

```
apply   (simp (no_asm) del: peek_and_def2)
apply   (tactic "ax_tac 1")
prefer 2
apply   (rule ax_subst_Val_allI)
apply    (tactic {* inst1_tac "P'4" "λaa v. Normal (?QQ aa v←?y)" *})
apply    (simp del: peek_and_def2)
apply    (tactic "ax_tac 1")
apply   (tactic "ax_tac 1")
apply  (tactic "ax_tac 1")
apply  (tactic "ax_tac 1")

apply (simp (no_asm))

apply (rule_tac Q' = "Normal ((λY (x, s) Z. arr_inv s ∧ (∃a. the (locals
s (Inl e)) = Addr a ∧ obj_class (the (globs s (Inl a))) = Ext)) ∧. initd
Ext ∧. heap_free 2)" in conseq2)
prefer 2
apply  clarsimp
apply (tactic "ax_tac 1")
apply (tactic "ax_tac 1")
defer
apply  (rule ax_subst_Var_allI)
apply  (tactic {* inst1_tac "P'16" "λu vf. Normal (?PP vf ∧. ?p) u" *})
apply  (simp (no_asm) del: split_paired_All peek_and_def2)
apply  (tactic "ax_tac 1" )
apply  (tactic "ax_tac 1" )

apply  (rule_tac Q' = "Normal ((λY s Z. arr_inv (snd s) ∧ vf=lvar (EName
e) (snd s)) ∧. heap_free 3 ∧. initd Ext)" in conseq2)
prefer 2
apply   (force elim!: arr_inv_new_obj atleast_free_SucD atleast_free_weaken)

apply  (rule ax_InitS)
apply     force
apply    (simp (no_asm))
apply   (rule ax_Init_Skip_lemma)
apply  (tactic {* simp_tac (simpset() delloop "split_all_tac") 1 *})
apply  (rule ax_InitS [THEN conseq1] )
apply      force
apply     (simp (no_asm))
apply    (unfold arr_viewed_from_def)
apply    (rule allI)
apply    (rule_tac P' = "Normal ?P" in conseq1)
apply     (tactic "ax_tac 1")
apply     (tactic "ax_tac 1")
apply     (rule_tac [2] ax_subst_Var_allI)
apply      (tactic {* inst1_tac "P'50" "λvf l vfa. Normal (?P vf l vfa)"
*})
apply     (tactic {* simp_tac (simpset() delloop "split_all_tac" delsimps
```

```
[split_paired_All, thm "peek_and_def2"]) 2 *})
apply      (tactic "ax_tac 2" )
apply       (tactic "ax_tac 3" )
apply       (tactic "ax_tac 3")
apply      (tactic {* inst1_tac "P" "λvf l vfa. Normal (?P vf l vfa←•)"
*})
apply      (tactic {* simp_tac (simpset() delloop "split_all_tac") 2 *})
apply      (tactic "ax_tac 2")
apply     (tactic "ax_tac 1" )
apply      (tactic "ax_tac 2" )
apply     (rule ax_derivs.Done [THEN conseq1])
apply     (tactic {* inst1_tac "Q35" "λvf. Normal ((λY s Z. vf=lvar (EName
e) (snd s)) ∧. heap_free 4 ∧. initd Base ∧. initd Ext)" *})
apply     (clarsimp split del: split_if)
apply     (frule atleast_free_weaken [THEN atleast_free_weaken])
apply     (drule initedD)
apply     (clarsimp elim!: atleast_free_SucD simp add: arr_inv_def)
apply    force
apply   (rule ax_triv_Init_Object [THEN peek_and_forget2, THEN conseq1])
apply     (rule wf_tprg)
apply    clarsimp
apply   (tactic {* inst1_tac "P35" "λvf. Normal ((λY s Z. vf = lvar (EName
e) (snd s)) ∧. heap_free 4 ∧. initd Ext)" *})
apply   clarsimp
apply  (tactic {* inst1_tac "PP" "λvf. Normal ((λY s Z. vf = lvar (EName
e) (snd s)) ∧. heap_free 4 ∧. Not ∘ initd Base)" *})
apply  clarsimp

apply (rule conseq1)
apply (tactic "ax_tac 1")
apply clarsimp
done


lemma Loop_Xcpt_benchmark:
 "Q = (λY (x,s) Z. x ≠ None ⟶ the_Bool (the (locals s i))) ⟹
  G,({}::'a triple set)⊢{Normal (λY s Z::'a. True)}
  .While(Lit (Bool True)) (If(Acc (LVar i)) (Throw (Acc (LVar xcpt)))
Else
     (Expr (Ass (LVar i) (Acc (LVar j))))). {Q}"
apply (rule_tac P' = "Q" and Q' = "Q←=False↓=•" in conseq12)
apply  safe
apply  (tactic "ax_tac 1" )
apply   (rule ax_Normal_cases)
prefer 2
apply    (rule ax_derivs.Xcpt [THEN conseq1], clarsimp simp add: Let_def)
apply   (rule conseq1)
apply    (tactic "ax_tac 1")
apply   clarsimp
```

```
prefer 2
apply  clarsimp
apply (tactic "ax_tac 1" )
apply  (tactic
  {* inst1_tac "P'20" "Normal (λs.. (λY s Z. True)↓=Val (the (locals
s i)))" *})
apply  (tactic "ax_tac 1")
apply  (rule conseq1)
apply   (tactic "ax_tac 1")
apply  clarsimp
apply (rule allI)
apply (rule ax_escape)
apply auto
apply  (rule conseq1)
apply   (tactic "ax_tac 1" )
apply   (tactic "ax_tac 1")
apply   (tactic "ax_tac 1")
apply  clarsimp
apply (rule_tac Q' = "Normal (λY s Z. True)" in conseq2)
prefer 2
apply  clarsimp
apply (rule conseq1)
apply  (tactic "ax_tac 1")
apply  (tactic "ax_tac 1")
prefer 2
apply   (rule ax_subst_Var_allI)
apply   (tactic {* inst1_tac "P'37" "λb Y ba Z vf. λY (x,s) Z. x=None
∧ snd vf = snd (lvar i s)" *})
apply   (rule allI)
apply   (rule_tac P' = "Normal ?P" in conseq1)
prefer 2
apply    clarsimp
apply   (tactic "ax_tac 1")
apply   (rule conseq1)
apply    (tactic "ax_tac 1")
apply   clarsimp
apply  (tactic "ax_tac 1")
apply clarsimp
done

end



theory AxSound = AxSem:
```

76 validity

consts

```
 triple_valid2:: "prog ⇒ nat ⇒        'a triple  ⇒ bool"
                                              (   "_|=_::_"[61,0, 58]
57)
    ax_valids2:: "prog ⇒ 'a triples ⇒ 'a triples ⇒ bool"
                                              ("_,_|=::_" [61,58,58]
57)
```

defs *triple_valid2_def: "G|=n::t ≡ case t of {P} t≻ {Q} ⇒*
∀Y s Z. P Y s Z ⟶ (∀L. s::≼(G,L) ⟶ (∀T. (normal s ⟶ (G,L)⊢t::T)
⟶
(∀Y' s'. G⊢s −t≻−n→ (Y',s') ⟶ Q Y' s' Z ∧ s'::≼(G,L))))"
defs *ax_valids2_def: "G,A|=::ts ≡ ∀n. (∀t∈A. G|=n::t) ⟶ (∀t∈ts.*
G|=n::t)"

lemma *triple_valid2_def2: "G|=n::{P} t≻ {Q} =*
(∀Y s Z. P Y s Z ⟶ (∀Y' s'. G⊢s −t≻−n→ (Y',s')⟶
(∀L. s::≼(G,L) ⟶ (∀T. (normal s ⟶ (G,L)⊢t::T) ⟶
Q Y' s' Z ∧ s'::≼(G,L))))"
apply *(unfold triple_valid2_def)*
apply *(simp (no_asm) add: split_paired_All)*
apply *blast*
done

lemma *triple_valid2_eq [rule_format (no_asm)]:*
"wf_prog G ==> triple_valid2 G = triple_valid G"
apply *(rule ext)*
apply *(rule ext)*
apply *(rule triple.induct)*
apply *(simp (no_asm) add: triple_valid_def2 triple_valid2_def2)*
apply *(rule iffI)*
apply *fast*
apply *clarify*
apply *(tactic "smp_tac 3 1")*
apply *(case_tac "normal s")*
apply *clarsimp*
apply *(blast dest: evaln_eval eval_type_sound [THEN conjunct1])*
apply *clarsimp*
done

lemma *ax_valids2_eq: "wf_prog G ⟹ G,A|=::ts = G,A|=ts"*
apply *(unfold ax_valids_def ax_valids2_def)*
apply *(force simp add: triple_valid2_eq)*
done

lemma *triple_valid2_Suc [rule_format (no_asm)]: "G|=Suc n::t ⟶ G|=n::t"*

```
apply (induct_tac "t")
apply (subst triple_valid2_def2)
apply (subst triple_valid2_def2)
apply (fast intro: evaln_nonstrict_Suc)
done

lemma Methd_triple_valid2_0: "G|=0::{Normal P} Methd C sig-> {Q}"
apply (clarsimp elim!: evaln_elim_cases simp add: triple_valid2_def2)
done

lemma Methd_triple_valid2_SucI:
"⟦G|=n::{Normal P} body G C sig->{Q}⟧
  ⟹ G|=Suc n::{Normal P} Methd C sig-> {Q}"
apply (simp (no_asm_use) add: triple_valid2_def2)
apply (intro strip, tactic "smp_tac 3 1", clarify)
apply (erule wt_elim_cases, erule evaln_elim_cases)
apply (unfold body_def Let_def)
apply clarsimp
apply blast
done

lemma triples_valid2_Suc: "Ball ts (triple_valid2 G (Suc n)) ⟹ Ball
ts (triple_valid2 G n)"
apply (fast intro: triple_valid2_Suc)
done

lemma "G|=n:insert t A = (G|=n:t ∧ G|=n:A)"
oops
```

77 soundness

```
lemma Methd_sound:
"⟦G,A∪  {{P} Methd-> {Q} | ms}|=::{{P} body G-> {Q} | ms}⟧ ⟹
  G,A|=::{{P} Methd-> {Q} | ms}"
apply (unfold ax_valids2_def mtriples_def)
apply (rule allI)
apply (induct_tac "n")
apply  (clarify, tactic {* pair_tac "x" 1 *}, simp (no_asm))
apply  (fast intro: Methd_triple_valid2_0)
apply (clarify, tactic {* pair_tac "xa" 1 *}, simp (no_asm))
apply (drule triples_valid2_Suc)
apply (erule (1) notE impE)
apply (drule_tac x = na in spec)
apply (tactic {* auto_tac (claset() addSIs [thm "Methd_triple_valid2_SucI"],
   simpset() addsimps [ball_Un] addloop ("split_all_tac", split_all_tac))
*})
done
```

```
lemma validsΩ2_inductI: "∀s t n Y' s'. G⊢s-t≻-n→ (Y',s') ⟶ t =
c ⟶
  Ball A (triple_valid2 G n) ⟶ (∀Y Z. P Y s Z ⟶
  (∀L. s::≼(G,L) ⟶ (∀T. (normal s ⟶ (G,L)⊢t::T) ⟶
  Q Y' s' Z ∧ s'::≼(G, L)))) ⟹
  G,A|⊨::{ {P} c≻ {Q}}"
apply (simp (no_asm) add: ax_validsΩ2_def triple_valid2_defΩ2)
apply clarsimp
done

ML_setup {*
Delsimprocs [evaln_expr_proc,evaln_var_proc,evaln_exprs_proc,evaln_stmt_proc]
*}

lemma Loop_sound: "⟦G,A|⊨::{ {P} e-≻ {P'}};
       G,A|⊨::{ {Normal (P'←=True)} .c. {P}}⟧ ⟹
       G,A|⊨::{ {P} .While(e) c. {(P'←=False)↓=•}}"
apply (rule validsΩ2_inductI)
apply ((rule allI)+, rule impI, tactic {* pair_tac "s" 1*}, tactic {*
pair_tac "s'" 1*})
apply (erule evaln.induct)
apply  simp_all
apply  clarify
apply  (erule_tac V = "G,A|⊨::{ {?P'} .c. {?P}}" in thin_rl)
apply  (simp_all (no_asm_use) add: ax_validsΩ2_def triple_valid2_defΩ2)
apply  (tactic "smp_tac 1 1", tactic "smp_tac 3 1", force)
apply clarify
apply (rule wt_elim_cases, assumption)
apply (tactic "smp_tac 1 1", tactic "smp_tac 1 1", tactic "smp_tac 3 1",

       tactic "smp_tac 2 1", tactic "smp_tac 1 1")
apply (erule impE, simp (no_asm), erule exI)
apply (simp add: imp_conjL split_tupled_all split_paired_All)
apply (case_tac "the_Bool b")
apply  clarsimp
apply  (erule_tac V = "c = While(e) c ⟶ ?P" in thin_rl)
apply  (case_tac "a")
apply   (simp_all)
apply   blast+
done

declare subst_Bool_defΩ2 [simp del]
lemma all_empty: "(!x. P) = P"
by simp
lemma sound_valid2_lemma:
"⟦∀v n. Ball A (triple_valid2 G n) ⟶ P v n; Ball A (triple_valid2 G
n)⟧⟹P v n"
by blast
ML {*
```

```
val fullsimptac = full_simp_tac(simpset() delsimps [thm "all_empty"]);
val sound_prepare_tac = EVERY'[REPEAT o thin_tac "?x ∈ ax_derivs G",
 full_simp_tac (simpset()addsimps[thm "ax_valids2_def",thm "triple_valid2_def2",
                                  thm "imp_conjL"] delsimps[thm "all_empty"]),
 Clarify_tac];
val sound_elim_tac = EVERY'[eresolve_tac (thms "evaln_elim_cases"),
        TRY o eresolve_tac (thms "wt_elim_cases"), fullsimptac, Clarify_tac];
val sound_valid2_tac = REPEAT o FIRST'[smp_tac 1,
                  datac (thm "sound_valid2_lemma") 1];
val sound_forw_hyp_tac = EVERY'[smp_tac 3 ORELSE'
 EVERY'[dtac spec,dtac spec,dtac spec,etac impE, Fast_tac],
 fullsimptac, smp_tac 2,TRY o smp_tac 1,
 TRY o EVERY'[etac impE, TRY o rtac impI, atac ORELSE' etac exI,
 fullsimptac, Clarify_tac, TRY o smp_tac 1]]
*}

lemma Call_sound:
 "⟦wf_prog G; G,A|⊨::{ {Normal P} e-≻ {Q}}; ∀a. G,A|⊨::{ {Q←Val a} ps≐≻
{R a}};
     ∀a vs D l. G,A|⊨::{ {(R a←Vals vs ∧.
                (λs. D = target mode (snd s) a cT ∧ l = locals (snd s))
;.
                init_lvars G D (mn,pTs) mode a vs) ∧.
                (λs. normal s ⟶ G⊢mode→D≼t)}
                Methd D (mn,pTs)-≻ {set_lvars l .; S}}⟧ ⟹
  G,A|⊨::{ {Normal P} {t,cT,mode}e..mn({pTs}ps)-≻ {S}}"
apply (tactic "EVERY'[sound_prepare_tac, sound_elim_tac, sound_valid2_tac]
1")
apply (rename_tac x1 s1 x2 s2 ab bb v vs m pTsa pns rT)
apply (tactic "smp_tac 5 1")
apply (tactic "sound_forw_hyp_tac 1")
apply (tactic "sound_forw_hyp_tac 1")
apply (drule max_spec2mheads)
apply (drule evaln_eval, drule (3) eval_ts)
apply (drule evaln_eval, frule (3) evals_ts)
apply (drule spec,drule spec,drule spec,erule impE,rule exI,blast)
apply (case_tac "if m then x2 else (np a') x2")
defer 1
apply  (rename_tac x, subgoal_tac "(Some x, s2)::≼(G, L)" )
prefer 2
apply   (force split add: split_if_asm)
apply  (simp del: if_raise_if_None)
apply  (tactic "smp_tac 2 1")
apply  (clarsimp simp add: init_lvars_def2)
apply  (drule spec,erule swap,erule conforms_set_locals [OF _ lconf_empty])
apply clarsimp
apply (drule Null_staticD)
apply (drule eval_gext', drule (1) conf_gext, frule (3) DynT_propI)
apply (erule (1) notE impE, tactic "smp_tac 2 1")
```

```
apply (tactic {* exhaust_cmethd_tac "the (cmethd G (target (invmode m
e) s2 a' cT) (mn, pTs))" 1 *}, clarsimp)
apply (drule (4) DynT_mheadsD, rule HOL.refl)
apply (clarify, drule wf_mdeclD1, clarify)
apply (drule (10) conforms_init_lvars, tactic "smp_tac 1 1", clarsimp)
apply (erule_tac V = "?P ⟶ ?Q" in thin_rl, erule_tac V = "?P ⟶ ?Q"
in thin_rl,
       erule impE, force dest!: wt_MethdI)
apply (force dest!: evaln_eval eval_gext' elim: conforms_return
            del: impCE simp add: init_lvars_def2)
done

lemma Init_sound: "⟦wf_prog G; the (class G C) = (sc, si, fs, ms, ini);

     G,A|⊨::{ {Normal ((P ∧. Not ∘ initd C) ;. supd (init_class_obj
G C))}
             .(if C = Object then Skip else init sc). {Q}};
  ∀l. G,A|⊨::{ {Q ∧. (λs. l = locals (snd s)) ;. set_lvars empty}
           .ini. {set_lvars l .; R}}⟧ ⟹
     G,A|⊨::{ {Normal (P ∧. Not ∘ initd C)} .init C. {R}}"
apply (tactic "EVERY'[sound_prepare_tac, sound_elim_tac, sound_valid2_tac]
1")
apply (clarsimp simp add: split_paired_Ex)
apply (drule spec, drule spec, drule spec, erule impE)
apply  (erule_tac V = "All ?P" in thin_rl, fast)
apply clarsimp
apply (tactic "smp_tac 2 1", drule spec, erule impE,
       erule (3) conforms_init_class_obj)
apply (drule (1) wf_prog_cdecl)
apply (erule impE, erule_tac V = "All ?P" in thin_rl,
       force dest: wf_cdecl_supD split add: split_if)
apply clarify
apply (drule spec, drule spec, drule spec, erule impE, fast)
apply (simp (no_asm_use) del: empty_def2)
apply (tactic "smp_tac 2 1")
apply (drule spec, erule impE, erule conforms_set_locals, rule lconf_empty)
apply (erule impE, rule impI, erule wf_cdecl_wt_init)
apply clarsimp
apply (erule (1) conforms_return, force dest: evaln_eval eval_gext')
done

lemma all_conjunct2: "∀l. P' l ∧ P l ⟹ ∀l. P l"
by fast

lemma all4_conjunct2:
  "∀a vs D l. (P' a vs D l ∧ P a vs D l) ⟹ ∀a vs D l. P a vs D l"
by fast

lemmas sound_lemmas = Call_sound Init_sound Loop_sound Methd_sound
```

```
lemma ax_sound2: "wf_prog G ⟹ G,A|⊢ts ⟹ G,A|⊨::ts"
apply (erule ax_derivs.induct)
apply (tactic {* TRYALL (eresolve_tac (thms "sound_lemmas") THEN_ALL_NEW

    eresolve_tac [asm_rl, thm "all_conjunct2", thm "all4_conjunct2"])
*})
apply(tactic "COND (has_fewer_prems(30+1)) (ALLGOALS sound_prepare_tac)
no_tac")

apply        fast
apply       fast

apply     fast
apply    (tactic "smp_tac 3 1", clarify, tactic "smp_tac 1 1", frule evaln_eval,
case_tac"fst s",clarsimp simp add: eval_type_sound [THEN conjunct1],
clarsimp)
apply   (simp (no_asm_use) add: type_ok_def, drule evaln_eval,fast)
apply  force


apply (tactic {* ALLGOALS sound_elim_tac *})
apply (tactic {* ALLGOALS (asm_simp_tac (noAll_simpset()
                          delsimps [thm "all_empty"])) *})
apply (frule wf_ws_prog,  frule (3) ty_expr_is_type [THEN type_is_class,

          THEN cfield_defpl_is_class])
apply (frule_tac [4] wt_init_comp_ty)
apply (tactic "ALLGOALS sound_valid2_tac")
apply (tactic "TRYALL sound_forw_hyp_tac")
apply (tactic {* TRYALL (EVERY'[dtac spec, TRY o EVERY'[rotate_tac ~1,

  dtac spec], dtac conjunct2, smp_tac 1,
  TRY o dres_inst_tac [("P","P'")] (thm "subst_Bool_the_BoolI")]) *})
apply (frule_tac [13] x = x1 in conforms_NormI)

apply (tactic "ALLGOALS (FIRST'[sound_forw_hyp_tac, (* Cons, Comp *)
                               smp_tac 2(*for NewC*), K all_tac])")



apply (clarsimp simp add: fvar_def2 Let_def split add: split_if_asm)


apply (clarsimp simp add: avar_def2)


apply (simp (no_asm_simp))
apply (erule (2) halloc_conforms, simp, simp)
```

```
apply (erule (1) halloc_conforms, simp, simp)


apply (case_tac "aa")
prefer 2
apply  clarsimp
apply (drule evaln_eval)+
apply (frule (3) eval_ts)
apply clarsimp
apply (frule (3) evar_ts [THEN conjunct2])
apply (frule wf_ws_prog)
apply (drule (2) conf_widen)
apply (drule_tac "s1.0" = b in eval_gext')
apply (clarsimp simp add: assign_conforms_def)


apply (erule impE, rule impI, rule_tac x = "if the_Bool b then T1 else
T2" in
       exI, force split add: split_if)
apply assumption


apply (tactic "sound_forw_hyp_tac 1")


apply (force split add: split_if)


apply (drule evaln_eval, drule (3) eval_ts)
apply clarsimp
apply (drule (3) Throw_lemma)
apply clarsimp


apply (frule (1) sxalloc_type_sound)
apply (erule sxalloc_elim_cases2)
apply  (tactic "smp_tac 3 1")
apply  (clarsimp split add: option.split_asm)
apply (clarsimp split add: option.split_asm)
apply (tactic "smp_tac 1 1")
apply (simp only: split add: split_if_asm)
prefer 2
apply  (tactic "smp_tac 3 1", erule_tac V = "All ?P" in thin_rl, clarsimp)
apply (drule spec, erule_tac V = "All ?P" in thin_rl, drule spec, drule
spec,
       erule impE, force)
apply (frule (2) Try_lemma)
```

apply *clarsimp*
apply *(fast elim!: conforms_deallocL)*

apply *(case_tac "x1", force)*
apply *clarsimp*
apply *(drule evaln_eval, drule (4) Fin_lemma)*
done
declare *subst_Bool_def2 [simp]*

theorem *ax_sound:*
"wf_prog G ⟹ G,(A::'a triple set)|⊢(ts::'a triple set) ⟹ G,A|⊨ts"
apply *(subst ax_valids2_eq [symmetric])*
apply *assumption*
apply *(erule (1) ax_sound2)*
done

end

theory *AxCompl = AxSem:*

78 set of not yet initialzed classes

constdefs

nyinitcls :: "prog ⇒ state ⇒ tname set"
"nyinitcls G s ≡ {C. is_class G C ∧ ¬ initd C s}"

lemma *nyinitcls_subset_class: "nyinitcls G s ⊆ {C. is_class G C}"*
apply *(unfold nyinitcls_def)*
apply *fast*
done

lemmas *finite_nyinitcls [simp] =*
finite_is_class [THEN nyinitcls_subset_class [THEN finite_subset], standard]

lemma *card_nyinitcls_bound: "card (nyinitcls G s) ≤ card {C. is_class G C}"*
apply *(rule nyinitcls_subset_class [THEN finite_is_class [THEN card_mono]])*
done

lemma *nyinitcls_set_locals_cong [simp]:*
"nyinitcls G (x,set_locals l s) = nyinitcls G (x,s)"
apply *(unfold nyinitcls_def)*
apply *(simp (no_asm))*
done

lemma *nyinitcls_xcpt_cong [simp]: "nyinitcls G (f x, y) = nyinitcls G (x, y)"*
apply *(unfold nyinitcls_def)*
apply *(simp (no_asm))*
done

lemma *nyinitcls_xupd_cong [simp]: "!!s. nyinitcls G (xupd f s) = nyinitcls G s"*
apply *(unfold nyinitcls_def)*
apply *(simp (no_asm_simp) only: split_tupled_all)*
apply *(simp (no_asm))*
done

lemma *card_nyinitcls_xcpt_congE [elim!]:*
"card (nyinitcls G (x, s)) ≤ n ⟹ card (nyinitcls G (y, s)) ≤ n"
apply *(unfold nyinitcls_def)*
apply *auto*
done

lemma *nyinitcls_new_xcpt_var [simp]:*
"nyinitcls G (new_xcpt_var vn s) = nyinitcls G s"
apply *(unfold nyinitcls_def)*
apply *(induct_tac "s")*
apply *(simp (no_asm))*
done

lemma *nyinitcls_init_lvars [simp]:*
"nyinitcls G ((init_lvars G C sig mode a' pvs) s) = nyinitcls G s"
apply *(induct_tac "s")*
apply *(simp (no_asm) add: init_lvars_def2 split add: split_if)*
done

lemma *nyinitcls_emptyD: "⟦nyinitcls G s = {}; is_class G C⟧ ⟹ initd C s"*
apply *(unfold nyinitcls_def)*
apply *fast*
done

lemma *card_Suc_lemma: "⟦card (insert a A) ≤ Suc n; a∉A; finite A⟧ ⟹ card A ≤ n"*
apply *(rotate_tac 1)*
apply *clarsimp*
done

lemma *nyinitcls_le_SucD:*
"⟦card (nyinitcls G (x,s)) ≤ Suc n; ¬inited C (globs s); class G C=Some y⟧ ⟹

```
  card (nyinitcls G (x,init_class_obj G C s)) ≤ n"
apply (subgoal_tac
       "nyinitcls G (x,s) = insert C (nyinitcls G (x,init_class_obj G
C s))")
apply  clarsimp
apply  (erule thin_rl)
apply  (rule card_Suc_lemma [OF _ _ finite_nyinitcls])
apply   (auto dest!: not_initedD elim!:
             simp add: nyinitcls_def inited_def split add: split_if_asm)
done

ML {* bind_thm("inited_gext'",permute_prems 0 1 (thm "inited_gext"))
*}
lemma nyinitcls_gext: "snd s≤|snd s' ⟹ nyinitcls G s' ⊆ nyinitcls
G s"
apply (unfold nyinitcls_def)
apply (force dest!: inited_gext')
done

lemma card_nyinitcls_gext:
  "⟦snd s≤|snd s'; card (nyinitcls G s) ≤ n⟧⟹ card (nyinitcls G s')
≤ n"
apply (rule le_trans)
apply  (rule card_mono)
apply   (rule finite_nyinitcls)
apply  (erule nyinitcls_gext)
apply assumption
done
```

79 init_{le}

```
constdefs
  init_le :: "prog ⇒ nat ⇒ state ⇒ bool"            ("_⊢init≤_"  [51,51]
50)
 "G⊢init≤n ≡ λs. card (nyinitcls G s) ≤ n"

lemma init_le_def2 [simp]: "(G⊢init≤n) s = (card (nyinitcls G s)≤n)"
apply (unfold init_le_def)
apply auto
done

lemma All_init_leD: "∀n::nat. G,A⊢{P ∧. G⊢init≤n} t≻ {Q} ⟹ G,A⊢{P}
t≻ {Q}"
apply (drule spec)
apply (erule conseq1)
apply clarsimp
apply (rule card_nyinitcls_bound)
done
```

80 Most General Triples and Formulas

constdefs

```
  remember_init_state :: "state assn"                ("�doteq")
  "�doteq ≡ λY s Z. s = Z"
```

lemma *remember_init_state_def2 [simp]: "�doteq Y = op ="*
apply *(unfold remember_init_state_def)*
apply *(simp (no_asm))*
done

consts

```
  MGF ::"[state assn, term, prog] ⇒ state triple"   ("{_} _≻ {_→}"[3,65,3]62)
  MGFn::"[nat       , term, prog] ⇒ state triple" ("{=:_} _≻ {_→}"[3,65,3]62)
```

defs

```
  MGF_def:
  "{P} t≻ {G→} ≡ {P} t≻ {λY s' s. G⊢s −t≻→ (Y,s')}"

  MGFn_def:
  "{=:n} t≻ {G→} ≡ {�doteq ∧. G⊢init≤n} t≻ {G→}"
```

lemma *MGF_valid: "G,{}⊨{�doteq} t≻ {G→}"*
apply *(unfold MGF_def)*
apply *(force dest!: evaln_eval simp add: ax_valids_def triple_valid_def2)*
done

lemma *MGF_res_eq_lemma [simp]:*
"(∀Y' Y s. Y = Y' ∧ P s ⟶ Q s) = (∀s. P s ⟶ Q s)"
apply *auto*
done

lemma *MGFn_def2:*
"G,A⊢{=:n} t≻ {G→} = G,A⊢{�doteq ∧. G⊢init≤n}
t≻ {λY s' s. G⊢s −t≻→ (Y,s')}"
apply *(unfold MGFn_def MGF_def)*
apply *fast*
done

lemma *MGF_MGFn_iff: "G,A⊢{�doteq} t≻ {G→} = (∀n. G,A⊢{=:n} t≻ {G→})"*
apply *(simp (no_asm_use) add: MGFn_def2 MGF_def)*
apply *safe*
apply *(erule_tac [2] All_init_leD)*
apply *(erule conseq1)*

apply *clarsimp*
done

lemma *MGFnD:*
"G,A⊢{=:n} t≻ {G→} ⟹
G,A⊢{(λY' s' s. s' = s ∧ P s) ∧. G⊢init≤n}
t≻ {(λY' s' s. G⊢s−t≻→(Y',s') ∧ P s) ∧. G⊢init≤n}"
apply *(unfold init_le_def)*
apply *(simp (no_asm_use) add: MGFn_def2)*
apply *(erule conseq12)*
apply *clarsimp*
apply *(erule (1) eval_gext [THEN card_nyinitcls_gext])*
done
lemmas *MGFnD' = MGFnD [of _ _ _ _ "λx. True"]*

lemma *MGFNormalI: "G,A⊢{Normal ≐} t≻ {G→} ⟹*
G,(A::state triple set)⊢{≐::state assn} t≻ {G→}"
apply *(unfold MGF_def)*
apply *(rule ax_Normal_cases)*
apply *(erule conseq1)*
apply *clarsimp*
apply *(rule ax_derivs.Xcpt [THEN conseq1])*
apply *(clarsimp simp add: Let_def)*
done

lemma *MGFNormalD: "G,A⊢{≐} t≻ {G→} ⟹ G,A⊢{Normal ≐} t≻ {G→}"*
apply *(unfold MGF_def)*
apply *(erule conseq1)*
apply *clarsimp*
done

lemma *MGFn_NormalI:*
"G,(A::state triple set)⊢{Normal((λY' s' s. s'=s ∧ normal s) ∧. G⊢init≤n)}t≻

{λY s' s. G⊢s −t≻→ (Y,s')} ⟹ G,A⊢{=:n}t≻{G→}"
apply *(simp (no_asm_use) add: MGFn_def2)*
apply *(rule ax_Normal_cases)*
apply *(erule conseq1)*
apply *clarsimp*
apply *(rule ax_derivs.Xcpt [THEN conseq1])*
apply *(clarsimp simp add: Let_def)*
done

lemma *MGFn_free_wt:*
"(∃T L. (G,L)⊢t::T) ⟶ G,(A::state triple set)⊢{=:n} t≻ {G→} ⟹

G,A⊢{=:n} t≻ {G→}"
apply *(rule MGFn_NormalI)*
apply *(rule ax_free_wt)*

apply *(auto elim: conseq12 simp add: MGFn_def MGF_def)*
done

81 main lemmas

declare *fun_upd_apply [simp del]*
declare *splitI2 [rule del]*

ML_setup *{**
Delsimprocs [eval_expr_proc, eval_var_proc, eval_exprs_proc, eval_stmt_proc]
**}*
ML *{**
```
val eval_css = (claset() delrules [thm "eval.Xcpt"] addSIs (thms "eval.intros")

               delrules[thm "eval.Expr", thm "eval.Init", thm "eval.Try"]

               addIs   [thm "eval.Expr", thm "eval.Init"]
               addSEs[thm "eval.Try"] delrules[equalityCE],
               simpset() addsimps [split_paired_all,Let_def]
 addsimprocs [eval_expr_proc,eval_var_proc,eval_exprs_proc,eval_stmt_proc]);
val eval_Force_tac = force_tac eval_css;

val wt_prepare_tac = EVERY'[
    rtac (thm "MGFn_free_wt"),
    clarsimp_tac (claset() addSEs (thms "wt_elim_cases"), simpset())]
val compl_prepare_tac = EVERY'[rtac (thm "MGFn_NormalI"), Simp_tac]
val forw_hyp_tac = EVERY'[etac (thm "MGFnD'" RS thm "conseq12"), Clarsimp_tac]
val forw_hyp_eval_Force_tac =
        EVERY'[TRY o rtac allI, forw_hyp_tac, eval_Force_tac]
```
**}*

lemma *MGFn_Init: "∀m. Suc m≤n ⟶ (∀t. G,A⊢{=:m} t≻ {G→}) ⟹*
* G,(A::state triple set)⊢{=:n} In1r (init C)≻ {G→}"*
apply *(tactic "wt_prepare_tac 1")*

apply *(tactic "compl_prepare_tac 1")*
apply *(rule_tac C = "initd C"* **in** *ax_cases)*
apply *(rule ax_derivs.Done [THEN conseq1])*
apply *(clarsimp intro!: init_done)*
apply *(rule_tac y = n* **in** *nat.exhaust, clarsimp)*
apply *(rule ax_impossible [THEN conseq1])*
apply *(force dest!: nyinitcls_emptyD)*
apply *clarsimp*
apply *(drule_tac x = "nat"* **in** *spec)*
apply *clarsimp*
apply *(rule_tac Q = " (λY s' (x,s) . G⊢ (x,init_class_obj G C s) − (if*
C=Object then Skip else init (fst (the (class G C))))→ s' ∧ x=None ∧
¬inited C (globs s)) ∧. G⊢init≤nat" **in** *surjective_pairing5 [THEN ax_derivs.Init])*
apply *(rule_tac P' = "Normal ((λY s' s. s' = supd (init_class_obj G*

```
C) s ∧ normal s ∧ ¬ initd C s) ∧. G⊢init≤nat) " in conseq1)
prefer 2
apply   (force elim!: nyinitcls_le_SucD)
apply  (simp split add: split_if, rule conjI, clarify)
apply   (rule ax_derivs.Skip [THEN conseq1], tactic "eval_Force_tac 1")
apply  clarify
apply  (drule spec)
apply  (erule MGFnD' [THEN conseq12])
apply  (tactic "force_tac (claset(), simpset() addsimprocs[eval_stmt_proc])
1")
apply (rule allI)
apply (drule spec)
apply (erule MGFnD' [THEN conseq12])
apply clarsimp
apply (tactic {* pair_tac "sa" 1 *})
apply (tactic"clarsimp_tac (claset(), simpset() addsimprocs[eval_stmt_proc])
1")
apply (rule eval_Init, force+)
done
lemmas MGFn_InitD = MGFn_Init [THEN MGFnD, THEN ax_NormalD]

lemma MGFn_Call:
"⟦∀C sig. G,(A::state triple set)⊢{=:n} In1l (Methd C sig)≻ {G→};
  G,A⊢{=:n} In1l e≻ {G→}; G,A⊢{=:n} In3 ps≻ {G→}⟧ ⟹
  G,A⊢{=:n} In1l ({t,cT,mode}e..mn({pTs'}ps))≻ {G→}"
apply (tactic "wt_prepare_tac 1")
apply (tactic "compl_prepare_tac 1")
apply (rule_tac R = "λa'. (λY (x2,s2) (x,s) . x = None ∧ (∃s1 pvs. G⊢Norm
s −e-≻a'→ s1 ∧ Y = In3 pvs ∧ G⊢s1 −ps≐≻pvs→ (x2,s2))) ∧. G⊢init≤n"
in ax_derivs.Call)
apply  (erule MGFnD [THEN ax_NormalD])
apply safe
apply  (erule_tac V = "All ?P" in thin_rl, tactic "forw_hyp_eval_Force_tac
1")
apply (drule spec, drule spec)
apply (erule MGFnD' [THEN conseq12])
apply (tactic "clarsimp_tac eval_css 1")
apply (erule (1) eval_Call)
apply   (rule HOL.refl)
apply  (simp (no_asm_simp))+
done

lemma MGF_altern: "G,A⊢{Normal (≐ ∧. p)} t≻ {G→} =
 G,A⊢{Normal ((λY s Z. ∀w s'. G⊢s −t≻→ (w,s') ⟶ (w,s') = Z) ∧.
p)}
  t≻ {λY s Z. (Y,s) = Z}"
apply (unfold MGF_def)
apply (auto del: conjI elim!: conseq12)
apply (case_tac "∃w s. G⊢Norm sa −t≻→ (w,s) ")
```

```
apply  (fast dest: unique_eval)
apply clarsimp
apply (erule thin_rl)
apply (erule thin_rl)
apply (drule split_paired_All [THEN subst])
apply (clarsimp elim!: state_not_single)
done


lemma MGFn_Loop:
"⟦G,(A::state triple set)⊢{=:n} In1l expr≻ {G→}; G,A⊢{=:n} In1r stmt≻
{G→}⟧ ⟹
  G,A⊢{=:n} In1r (While(expr) stmt)≻ {G→}"
apply (rule MGFn_NormalI, simp)
apply (rule_tac p2 = "λs. card (nyinitcls G s) ≤ n" in
          MGF_altern [unfolded MGF_def, THEN iffD2, THEN conseq1])
prefer 2
apply  clarsimp
apply (rule_tac P' = "((λY s Z. ∀w s'. G⊢s −In1r (While(expr) stmt)
≻→ (w,s') ⟶ (w,s') = Z) ∧. (λs. card (nyinitcls G s) ≤ n))" in conseq12)
prefer 2
apply  clarsimp
apply  (tactic "smp_tac 1 1", erule_tac V = "All ?P" in thin_rl)
apply  (rule_tac [2] P' = " (λb s (Y',s') . (∃s0. G⊢s0 −In1l exprs≻→
(b,s)) ∧ (if the_Bool (the_In1 b) then (∀s'' w s0. G⊢s −stmt→ s'' ∧
G⊢s'' −In1r (While(expr) stmt) ≻→ (w,s0) ⟶ (w,s0) = (Y',s')) else
(•,s) = (Y',s'))) ∧. G⊢init≤n" in polymorphic_Loop)
apply   (force dest!: eval.Loop split add: split_if_asm)
prefer 2
apply  (erule MGFnD' [THEN conseq12])
apply  clarsimp
apply  (erule_tac V = "card (nyinitcls G s') ≤ n" in thin_rl)
apply  (tactic "eval_Force_tac 1")
apply (erule MGFnD' [THEN conseq12] , clarsimp)
apply (rule conjI, erule exI)
apply (tactic "clarsimp_tac eval_css 1")
apply (case_tac "a")
prefer 2
apply  (force split add: split_if)
apply (clarsimp split add: split_if)
apply (rule conjI, (tactic {* force_tac (claset() addSDs [thm "eval.Loop"],
  simpset() addsimps [split_paired_all] addsimprocs [eval_stmt_proc])
1*})+)
done


lemma MGFn_lemma [rule_format (no_asm)]:
 "∀n C sig. G,(A::state triple set)⊢{=:n} In1l (Methd C sig)≻ {G→}
⟹
```

```
  ∀t. G,A⊢{=:n} t≻ {G→}"
apply (rule full_nat_induct)
apply (rule allI)
apply (drule_tac x = n in spec)
apply (drule_tac psi = "All ?P" in asm_rl)
apply (subgoal_tac "∀v e c es. G,A⊢{=:n} In2 v≻ {G→} ∧ G,A⊢{=:n} In1l
e≻ {G→} ∧ G,A⊢{=:n} In1r c≻ {G→} ∧ G,A⊢{=:n} In3 es≻ {G→}")
apply  (tactic "Clarify_tac 2")
apply  (induct_tac "t")
apply    (induct_tac "a")
apply     fast+
apply (rule var_expr_stmt.induct)

prefer 14 apply fast
prefer 13 apply (erule (2) MGFn_Call)
apply (erule_tac [!] V = "All ?P" in thin_rl)
apply (erule_tac [22] MGFn_Init)
prefer 18 apply (erule (1) MGFn_Loop)
apply (tactic "ALLGOALS compl_prepare_tac")

apply (rule ax_derivs.LVar [THEN conseq1], tactic "eval_Force_tac 1")

apply (rule ax_derivs.FVar)
apply  (erule MGFn_InitD)
apply (tactic "forw_hyp_eval_Force_tac 1")

apply (rule ax_derivs.AVar)
apply  (erule MGFnD [THEN ax_NormalD])
apply (tactic "forw_hyp_eval_Force_tac 1")

apply (rule ax_derivs.NewC)
apply (erule MGFn_InitD [THEN conseq2])
apply (tactic "eval_Force_tac 1")

apply (rule_tac Q = "(λY' s' s. normal s ∧ G⊢s −In1r (init_comp_ty ty)
≻→ (Y',s')) ∧. G⊢init≤n" in ax_derivs.NewA)
apply  (simp add: init_comp_ty_def split add: split_if)
apply   (rule conjI, clarsimp)
apply   (erule MGFn_InitD [THEN conseq2])
apply   (tactic "clarsimp_tac eval_css 1")
apply  clarsimp
apply  (rule ax_derivs.Skip [THEN conseq1], tactic "eval_Force_tac 1")
apply (tactic "forw_hyp_eval_Force_tac 1")

apply (erule MGFnD'[THEN conseq12,THEN ax_derivs.Cast],tactic"eval_Force_tac
1")

apply (erule MGFnD'[THEN conseq12,THEN ax_derivs.Inst],tactic"eval_Force_tac
1")
```

```
apply (rule ax_derivs.Lit [THEN conseq1], tactic "eval_Force_tac 1")
apply (rule ax_derivs.Super [THEN conseq1], tactic "eval_Force_tac 1")
apply (erule MGFnD'[THEN conseq12,THEN ax_derivs.Acc],tactic"eval_Force_tac
1")

apply (rule ax_derivs.Ass)
apply  (erule MGFnD [THEN ax_NormalD])
apply (tactic "forw_hyp_eval_Force_tac 1")

apply (rule ax_derivs.Cond)
apply  (erule MGFnD [THEN ax_NormalD])
apply (rule allI)
apply (rule ax_Normal_cases)
prefer 2
apply  (rule ax_derivs.Xcpt [THEN conseq1], clarsimp simp add: Let_def)
apply  (tactic "eval_Force_tac 1")
apply (case_tac "b")
apply  (simp, tactic "forw_hyp_eval_Force_tac 1")
apply (simp, tactic "forw_hyp_eval_Force_tac 1")

apply (rule_tac R = " (λY' s' s. normal s ∧ (∃s''. G⊢s −init tname→
s'' ∧ G⊢s'' −stmt→ s')) ∧. G⊢init≤n" in ax_derivs.Body)
apply   (erule MGFn_InitD)
apply  (tactic "forw_hyp_eval_Force_tac 1")
apply (tactic "forw_hyp_eval_Force_tac 1")

apply (rule ax_derivs.Skip [THEN conseq1], tactic "eval_Force_tac 1")

apply (erule MGFnD'[THEN conseq12,THEN ax_derivs.Expr],tactic"eval_Force_tac
1")

apply (rule ax_derivs.Comp)
apply  (erule MGFnD [THEN ax_NormalD])
apply (tactic "forw_hyp_eval_Force_tac 1")

apply (rule ax_derivs.If)
apply  (erule MGFnD [THEN ax_NormalD])
apply (rule allI)
apply (rule ax_Normal_cases)
prefer 2
apply  (rule ax_derivs.Xcpt [THEN conseq1], clarsimp simp add: Let_def)
apply  (tactic "eval_Force_tac 1")
apply (case_tac "b")
apply  (simp, tactic "forw_hyp_eval_Force_tac 1")
apply (simp, tactic "forw_hyp_eval_Force_tac 1")

apply (erule MGFnD' [THEN conseq12, THEN ax_derivs.Throw])
apply (tactic "clarsimp_tac eval_css 1")
```

```
apply (rule_tac Q = " (λY' s' s. normal s ∧ (∃s''. G⊢s −In1r stmt1≻→
(Y',s'') ∧ G⊢s'' −sxalloc→ s')) ∧. G⊢init≤n" in ax_derivs.Try)
apply   (tactic "eval_Force_tac 3")
apply  (tactic "forw_hyp_eval_Force_tac 2")
apply (erule MGFnD [THEN ax_NormalD, THEN conseq2])
apply (tactic "clarsimp_tac eval_css 1")
apply (force elim: sxalloc_gext [THEN card_nyinitcls_gext])

apply (rule_tac Q = " (λY' s' s. normal s ∧ G⊢s −stmt1→ s') ∧. G⊢init≤n"
in ax_derivs.Fin)
apply  (tactic "forw_hyp_eval_Force_tac 1")
apply (rule allI)
apply (tactic "forw_hyp_tac 1")
apply (tactic {* pair_tac "sb" 1 *})
apply (tactic"clarsimp_tac (claset(),simpset() addsimprocs [eval_stmt_proc])
1")
apply (drule (1) eval.Fin)
apply clarsimp

apply (rule ax_derivs.Nil [THEN conseq1], tactic "eval_Force_tac 1")

apply (rule ax_derivs.Cons)
apply  (erule MGFnD [THEN ax_NormalD])
apply (tactic "forw_hyp_eval_Force_tac 1")
done

lemma MGF_asm: "∀C sig. is_methd G C sig ⟶ G,A⊢{�doteq} In1l (Methd C
sig)≻ {G→} ⟹
  G,(A::state triple set)⊢{�doteq} t≻ {G→}"
apply (simp (no_asm_use) add: MGF_MGFn_iff)
apply (rule allI)
apply (rule MGFn_lemma)
apply (intro strip)
apply (rule MGFn_free_wt)
apply (fast dest: wt_Methd_is_methd)
done

declare splitI2 [intro!]
ML_setup {*
Addsimprocs [ eval_expr_proc, eval_var_proc, eval_exprs_proc, eval_stmt_proc]
*}
```

82 nested version

```
lemma nesting_lemma' [rule_format (no_asm)]: "[| !!A ts. ts <= A ==>
P A ts;
  !!A pn. !b:bdy pn. P (insert (mgf_call pn) A) {mgf b} ==> P A {mgf_call
pn};
  !!A t. !pn:U. P A {mgf_call pn} ==> P A {mgf t};
```

```
         finite U; uA = mgf_call`U |] ==>
  !A. A <= uA --> n <= card uA --> card A = card uA - n --> (!t. P A {mgf
t})"
proof -
  assume ax_derivs_asm:    "!!A ts. ts <= A ==> P A ts"
  assume MGF_nested_Methd: "!!A pn. !b:bdy pn. P (insert (mgf_call pn)
A)
                                                  {mgf b} ==> P A {mgf_call
pn}"
  assume MGF_asm:          "!!A t. !pn:U. P A {mgf_call pn} ==> P A {mgf
t}"
  assume "finite U" "uA = mgf_call`U"
  then show ?thesis
    apply -
    apply (induct_tac "n")
    apply  (tactic "ALLGOALS Clarsimp_tac")
    apply  (tactic "dtac (permute_prems 0 1 card_seteq) 1")
    apply    simp
    apply   (erule finite_imageI)
    apply  (simp add: MGF_asm ax_derivs_asm)
    apply (rule MGF_asm)
    apply (rule ballI)
    apply (case_tac "mgf_call pn : A")
    apply  (fast intro: ax_derivs_asm)
    apply (rule MGF_nested_Methd)
    apply (rule ballI)
    apply (drule spec, erule impE, erule_tac [2] impE, erule_tac [3] impE,

           erule_tac [4] spec)
    apply   fast
    apply  (erule Suc_leD)
    apply (drule finite_subset)
    apply (erule finite_imageI)
    apply auto
    apply arith
  done
qed

lemma nesting_lemma [rule_format (no_asm)]: "[| !!A ts. ts <= A ==> P
A ts;
  !!A pn. !b:bdy pn. P (insert (mgf (f pn)) A) {mgf b} ==> P A {mgf (f
pn)};
          !!A t. !pn:U. P A {mgf (f pn)} ==> P A {mgf t};
          finite U |] ==> P {} {mgf t}"
proof -
  assume 2: "!!A pn. !b:bdy pn. P (insert (mgf (f pn)) A) {mgf b} ==>
P A {mgf (f pn)}"
  assume 3: "!!A t. !pn:U. P A {mgf (f pn)} ==> P A {mgf t}"
  assume "!!A ts. ts <= A ==> P A ts" "finite U"
```

```
  then show ?thesis
    apply -
    apply (rule_tac mgf = "mgf" in nesting_lemma')
    apply (erule_tac [2] 2)
    apply (rule_tac [2] 3)
    apply (rule_tac [6] le_refl)
    apply auto
  done
qed

lemma MGF_nested_Methd: "⟦
  G,insert ({Normal ≐} In1l (Methd  C sig) ≻{G→}) A⊢
           {Normal ≐} In1l (body G C sig) ≻{G→}
 ⟧ ⟹  G,A⊢{Normal ≐} In1l (Methd  C sig) ≻{G→}"
apply (unfold MGF_def)
apply (rule ax_MethdN)
apply (erule conseq2)
apply clarsimp
apply (erule MethdI)
done

lemma MGF_deriv: "ws_prog G ⟹ G,({}::state triple set)⊢{≐} t≻ {G→}"
apply (rule MGFNormalI)
apply (rule_tac mgf = "λt. {Normal ≐} t≻ {G→}" and
                bdy = "λ (C,sig) .{In1l (body G C sig) }" and
                f = "λ (C,sig) . In1l (Methd C sig) " in nesting_lemma)
apply    (erule ax_derivs.asm)
apply   (clarsimp simp add: split_tupled_all)
apply   (erule MGF_nested_Methd)
apply  (erule_tac [2] finite_is_methd)
apply (rule MGF_asm [THEN MGFNormalD])
apply clarify
apply (rule MGFNormalI)
apply force
done
```

83 simultaneous version

```
lemma MGF_simult_Methd_lemma: "finite ms ⟹
  G,A∪ (λ(C,sig). {Normal ≐} In1l (Methd  C sig)≻ {G→}) ‘ ms
     |⊢(λ(C,sig). {Normal ≐} In1l (body G C sig)≻ {G→}) ‘ ms ⟹
  G,A|⊢(λ(C,sig). {Normal ≐} In1l (Methd  C sig)≻ {G→}) ‘ ms"
apply (unfold MGF_def)
apply (rule ax_derivs.Methd [unfolded mtriples_def])
apply (erule ax_finite_pointwise)
prefer 2
apply  (rule ax_derivs.asm)
apply  fast
apply clarsimp
```

```
apply (rule conseq2)
apply  (erule (1) ax_methods_spec)
apply clarsimp
apply (erule eval_Methd)
done

lemma MGF_simult_Methd: "ws_prog G ⟹
   G,({}::state triple set)|⊢(λ(C,sig). {Normal �doteq} In1l (Methd C sig)≻
{G→})
   ‘ Collect (split (is_methd G)) "
apply (frule finite_is_methd)
apply (rule MGF_simult_Methd_lemma)
apply  assumption
apply (erule ax_finite_pointwise)
prefer 2
apply  (rule ax_derivs.asm)
apply  blast
apply clarsimp
apply (rule MGF_asm [THEN MGFNormalD])
apply clarify
apply (rule MGFNormalI)
apply force
done

lemma MGF_deriv: "ws_prog G ⟹ G,({}::state triple set)⊢{�doteq} t≻ {G→}"
apply (rule MGF_asm)
apply (intro strip)
apply (rule MGFNormalI)
apply (rule ax_derivs.weaken)
apply  (erule MGF_simult_Methd)
apply force
done
```

84 corollaries

```
lemma MGF_complete: "G,{}|=|{P} t≻ {Q} ⟹ G,({}::state triple set)⊢{�doteq}
t≻ {G→} ⟹
  G,({}::state triple set)⊢{P::state assn} t≻ {Q}"
apply (rule ax_no_hazard)
apply (unfold MGF_def)
apply (erule conseq12)
apply (simp (no_asm_use) add: ax_valids_def triple_valid_def)
apply (fast dest!: eval_evaln)
done

theorem ax_complete: "ws_prog G ⟹
  G,{}|=|{P::state assn} t≻ {Q} ⟹ G,({}::state triple set)⊢{P} t≻ {Q}"
apply (erule MGF_complete)
apply (erule MGF_deriv)
```

done

end