

# MiniML

Wolfgang Naraschewski and Tobias Nipkow

June 17, 2024

## Abstract

This theory defines the type inference rules and the type inference algorithm  $W$  for MiniML (simply-typed lambda terms with `let`) due to Milner. It proves the soundness and completeness of  $W$  w.r.t. the rules.

A report describing the theory is found in [1] and [2].

## 1 Universal error monad

```
theory Maybe
imports Main
begin
```

### definition

```
option_bind :: "[’a option, ’a => ’b option] => ’b option" where
"option_bind m f = (case m of None => None | Some r => f r)"
```

```
syntax "_option_bind" :: "[pttrns, ’a option, ’b] => ’c" ("(_ := _;//_)" 0)
translations "P := E; F" == "CONST option_bind E ( $\lambda$ P. F)"
```

— constructor laws for `option_bind`

```
lemma option_bind_Some: "option_bind (Some s) f = (f s)"
  by (simp add: option_bind_def)
```

```
lemma option_bind_None: "option_bind None f = None"
  by (simp add: option_bind_def)
```

```
declare option_bind_Some [simp] option_bind_None [simp]
```

— expansion of `option_bind`

```
lemma split_option_bind: "P(option_bind res f) =
  ((res = None  $\longrightarrow$  P None)  $\wedge$  ( $\forall$ s. res = Some s  $\longrightarrow$  P(f s)))"
  unfolding option_bind_def
  by (rule option.split)
```

```
lemma split_option_bind_asm: "P(option_bind res f) =
  ( $\sim$  ((res = None  $\wedge$   $\neg$  P None)  $\vee$  ( $\exists$ s. res = Some s  $\wedge$   $\neg$  P(f s))))"
```

```

unfolding option_bind_def
by (rule option.split_asm)

lemma option_bind_eq_None [simp]:
  "((option_bind m f) = None) = ((m=None) | (∃p. m = Some p ∧ f p = None))"
  by (simp split: split_option_bind)

lemma rotate_Some: "(y = Some x) = (Some x = y)"
  by (simp add: eq_sym_conv)

end

```

## 2 MiniML-types and type substitutions

```

theory Type
imports Maybe
begin

— type expressions
datatype "typ" = TVar nat | Fun "typ" "typ" (infixr "->" 70)

— type schemata
datatype type_scheme = FVar nat | BVar nat | SFun type_scheme type_scheme (infixr "=>"
70)

— embedding types into type schemata
fun mk_scheme :: "typ => type_scheme" where
  "mk_scheme (TVar n) = (FVar n)"
| "mk_scheme (t1 -> t2) = ((mk_scheme t1) ==> (mk_scheme t2))"

— type variable substitution
type_synonym subst = "nat => typ"

class type_struct =
  fixes free_tv :: "'a => nat set"
  — free_tv s: the type variables occurring freely in the type structure s
  fixes free_tv_ML :: "'a => nat list"
  — executable version of free_tv: Implementation with lists
  fixes bound_tv :: "'a => nat set"
  — bound_tv s: the type variables occurring bound in the type structure s
  fixes min_new_bound_tv :: "'a => nat"
  — minimal new free / bound variable
  fixes app_subst :: "subst => 'a => 'a" ("$$")
  — extension of substitution to type structures

instantiation "typ" :: type_struct
begin

fun free_tv_typ where

```

```

    free_tv_TVar:    "free_tv (TVar m) = {m}"
  | free_tv_Fun:    "free_tv (t1 -> t2) = (free_tv t1) Un (free_tv t2)"

fun app_subst_typ where
  app_subst_TVar: "$ S (TVar n) = S n"
  | app_subst_Fun: "$ S (t1 -> t2) = ($ S t1) -> ($ S t2)"

instance ..

end

instantiation type_scheme :: type_struct
begin

fun free_tv_type_scheme where
  "free_tv (FVar m) = {m}"
  | "free_tv (BVar m) = {}"
  | "free_tv (S1 ==> S2) = (free_tv S1) Un (free_tv S2)"

fun free_tv_ML_type_scheme where
  "free_tv_ML (FVar m) = [m]"
  | "free_tv_ML (BVar m) = []"
  | "free_tv_ML (S1 ==> S2) = (free_tv_ML S1) @ (free_tv_ML S2)"

fun bound_tv_type_scheme where
  "bound_tv (FVar m) = {}"
  | "bound_tv (BVar m) = {m}"
  | "bound_tv (S1 ==> S2) = (bound_tv S1) Un (bound_tv S2)"

fun min_new_bound_tv_type_scheme where
  "min_new_bound_tv (FVar n) = 0"
  | "min_new_bound_tv (BVar n) = Suc n"
  | "min_new_bound_tv (sch1 ==> sch2) = max (min_new_bound_tv sch1) (min_new_bound_tv sch2)"

fun app_subst_type_scheme where
  "$ S (FVar n) = mk_scheme (S n)"
  | "$ S (BVar n) = (BVar n)"
  | "$ S (sch1 ==> sch2) = ($ S sch1) ==> ($ S sch2)"

instance ..

end

instantiation list :: (type_struct) type_struct
begin

fun free_tv_list where
  "free_tv [] = {}"
  | "free_tv (x#l) = (free_tv x) Un (free_tv l)"

```

```

fun free_tv_ML_list where
  "free_tv_ML [] = []"
| "free_tv_ML (x#l) = (free_tv_ML x) @ (free_tv_ML l)"

```

```

fun bound_tv_list where
  "bound_tv [] = {}"
| "bound_tv (x#l) = (bound_tv x) Un (bound_tv l)"

```

```

definition app_subst_list where
  app_subst_list: "$ S = map ($ S)"

```

```

instance ..

```

```

end

```

`new_tv s n` computes whether `n` is a new type variable w.r.t. a type structure `s`, i.e. whether `n` is greater than any type variable occurring in the type structure

```

definition
  new_tv :: "[nat, 'a::type_struct] => bool" where
  "new_tv n ts = (∀m. m:(free_tv ts) → m<n)"

```

— identity

```

definition
  id_subst :: subst where
  "id_subst = (λn. TVar n)"

```

— domain of a substitution

```

definition
  dom :: "subst => nat set" where
  "dom S = {n. S n ≠ TVar n}"

```

— codomain of a substitution: the introduced variables

```

definition
  cod :: "subst => nat set" where
  "cod S = (UN m:dom S. (free_tv (S m)))"

```

```

class of_nat =
  fixes of_nat :: "nat ⇒ 'a"

```

```

instantiation nat :: of_nat
begin

```

```

definition
  "of_nat n = n"

```

```

instance ..

```

```

end

```

```

class typ_of =
  fixes typ_of :: "'a ⇒ typ"

instantiation "typ" :: typ_of
begin

definition
  "typ_of T = T"

instance ..

end

instantiation "fun" :: (of_nat, typ_of) type_struct
begin

definition free_tv_fun where
  "free_tv f = (let S = λn. typ_of (f (of_nat n)) in (dom S) Un (cod S))"

instance ..

end

lemma free_tv_subst:
  "free_tv S = (dom S) Un (cod S)"
by (simp add: free_tv_fun_def of_nat_nat_def typ_of_typ_def )

— unification algorithm mgu
axiomatization mgu :: "typ ⇒ typ ⇒ subst option" where
  mgu_eq: "mgu t1 t2 = Some U ⇒ $U t1 = $U t2"
  and mgu_mg: "[| (mgu t1 t2) = Some U; $S t1 = $S t2 |] ==> ∃R. S = $R ∘ U"
  and mgu_Some: "$S t1 = $S t2 ⇒ ∃U. mgu t1 t2 = Some U"
  and mgu_free: "mgu t1 t2 = Some U ⇒ free_tv U ⊆ free_tv t1 ∪ free_tv t2"

lemma mk_scheme_Fun:
  "mk_scheme t = sch1 ==> sch2 ⇒ (∃t1 t2. sch1 = mk_scheme t1 ∧ sch2 = mk_scheme t2)"
proof (induction t)
  case TVar thus ?case by auto
next
  case Fun thus ?case by auto
qed

lemma mk_scheme_injective: "(mk_scheme t = mk_scheme t') ⇒ t = t'"
proof (induction t arbitrary: t')
  case TVar thus ?case by (cases t') auto
next
  case Fun thus ?case by (cases t') auto

```

qed

```
lemma free_tv_mk_scheme[simp]: "free_tv (mk_scheme t) = free_tv t"
by (induction t) auto
```

```
lemma subst_mk_scheme[simp]: "$ S (mk_scheme t) = mk_scheme ($ S t)"
by (induction t) auto
```

— constructor laws for `app_subst`

```
lemma app_subst_Nil[simp]:
  "$ S [] = []"
by (simp add: app_subst_list)
```

```
lemma app_subst_Cons[simp]:
  "$ S (x#l) = ($ S x)#($ S l)"
by (simp add: app_subst_list)
```

— constructor laws for `new_tv`

```
lemma new_tv_TVar[simp]:
  "new_tv n (TVar m) = (m < n)"
by (simp add: new_tv_def)
```

```
lemma new_tv_FVar[simp]:
  "new_tv n (FVar m) = (m < n)"
by (simp add: new_tv_def)
```

```
lemma new_tv_BVar[simp]:
  "new_tv n (BVar m) = True"
by (simp add: new_tv_def)
```

```
lemma new_tv_Fun[simp]:
  "new_tv n (t1 -> t2) = (new_tv n t1 ∧ new_tv n t2)"
by (auto simp: new_tv_def)
```

```
lemma new_tv_Fun2[simp]:
  "new_tv n (t1 ==> t2) = (new_tv n t1 ∧ new_tv n t2)"
by (auto simp: new_tv_def)
```

```
lemma new_tv_Nil[simp]:
  "new_tv n []"
by (simp add: new_tv_def)
```

```
lemma new_tv_Cons[simp]:
  "new_tv n (x#l) = (new_tv n x ∧ new_tv n l)"
by (auto simp: new_tv_def)
```

```

lemma new_tv_TVar_subst [simp]: "new_tv n TVar"
by (simp add: new_tv_def free_tv_subst dom_def cod_def)

lemma new_tv_id_subst [simp]: "new_tv n id_subst"
by (simp add: id_subst_def new_tv_def free_tv_subst dom_def cod_def)

lemma new_if_subst_type_scheme: "new_tv n (sch::type_scheme)  $\implies$ 
 $\$(\lambda k. \text{if } k < n \text{ then } S \ k \text{ else } S' \ k) \text{ sch} = \$S \text{ sch}"$ 
by (induction sch) simp_all

lemma new_if_subst_type_scheme_list: "new_tv n (A::type_scheme list)  $\implies$ 
 $\$(\lambda k. \text{if } k < n \text{ then } S \ k \text{ else } S' \ k) \ A = \$S \ A"$ 
by (induction A) (simp_all add: new_if_subst_type_scheme)

— constructor laws for dom and cod

lemma dom_id_subst [simp]: "dom id_subst = {}"
unfolding dom_def id_subst_def empty_def by simp

lemma cod_id_subst [simp]: "cod id_subst = {}"
unfolding cod_def by simp

lemma free_tv_id_subst [simp]: "free_tv id_subst = {}"
unfolding free_tv_subst by simp

lemma free_tv_nth_A_impl_free_tv_A:
" $\llbracket n < \text{length } A; \ x : \text{free\_tv } (A!n) \rrbracket \implies x : \text{free\_tv } A"$ 
proof (induction A arbitrary: n)
case Nil thus ?case by simp
next
case (Cons a A)
then show ?case by (fastforce simp add: nth_Cons' split: if_splits)
qed

if two substitutions yield the same result if applied to a type structure the substitutions
coincide on the free type variables occurring in the type structure

lemma typ_substitutions_only_on_free_variables:
" $(\forall x \in \text{free\_tv } t. (S \ x) = (S' \ x)) \implies \$ S \ (t::\text{typ}) = \$ S' \ t"$ 
by (induct t) simp_all

lemma eq_free_eq_subst_te: " $(\forall n. n \in (\text{free\_tv } t) \longrightarrow S1 \ n = S2 \ n) \implies \$ S1 \ (t::\text{typ}) =$ 
 $\$ S2 \ t"$ 
apply (rule typ_substitutions_only_on_free_variables)
apply simp
done

```

```

lemma scheme_substitutions_only_on_free_variables:
  "( $\forall x \in \text{free\_tv } \text{sch}. (S x) = (S' x)$ )  $\implies$  $ S (sch::type_scheme) = $ S' sch"
  by (induct sch) simp_all

lemma eq_free_eq_subst_type_scheme:
  "( $\forall n. n \in (\text{free\_tv } \text{sch}) \longrightarrow S1 n = S2 n$ )  $\implies$  $ S1 (sch::type_scheme) = $ S2 sch"
  apply (rule scheme_substitutions_only_on_free_variables)
  apply simp
  done

lemma eq_free_eq_subst_scheme_list:
  "( $\forall n. n \in (\text{free\_tv } A) \longrightarrow S1 n = S2 n$ )  $\implies$  $S1 (A::type_scheme list) = $S2 A"
  proof (induct A)
    case Nil then show ?case by fastforce
  next
    case Cons then show ?case by (fastforce intro: eq_free_eq_subst_type_scheme)
  qed

lemma weaken_asm_Un: "( $(\forall x \in A. (P x)) \longrightarrow Q$ )  $\implies$  ( $(\forall x \in A \cup B. (P x)) \longrightarrow Q$ )"
  by fast

lemma scheme_list_substitutions_only_on_free_variables:
  "( $\forall x \in \text{free\_tv } A. (S x) = (S' x)$ )  $\implies$  $ S (A::type_scheme list) = $ S' A"
  by (induction A) (auto simp add: eq_free_eq_subst_type_scheme)

lemma eq_subst_te_eq_free:
  "$ S1 (t::typ) = $ S2 t  $\implies$  n:(free_tv t)  $\implies$  S1 n = S2 n"
  by (induct t) auto

lemma eq_subst_type_scheme_eq_free:
  "[[ $ S1 (sch::type_scheme) = $ S2 sch; n:(free_tv sch) ] ]  $\implies$  S1 n = S2 n"
  by (induction "sch") (auto dest: mk_scheme_injective)

lemma eq_subst_scheme_list_eq_free:
  "[[ $S1 (A::type_scheme list) = $S2 A; n:(free_tv A) ] ]  $\implies$  S1 n = S2 n"
  proof (induct A)
    case Nil
      then show ?case by fastforce
    next
      case Cons
        then show ?case by (fastforce intro: eq_subst_type_scheme_eq_free)
  qed

lemma codD: "v : cod S  $\implies$  v : free_tv S"
  unfolding free_tv_subst by blast

lemma not_free_impl_id: "x  $\notin$  free_tv S  $\implies$  S x = TVar x"
  unfolding free_tv_subst dom_def by blast

```



```

lemma free_tv_le_new_tv: "[| new_tv n t; m:free_tv t |] ==> m<n"
  unfolding new_tv_def by blast

lemma cod_app_subst:
  "[| v : free_tv(S n); v≠n |] ==> v : cod S"
  by (force simp add: dom_def cod_def UNION_eq Bex_def)

lemma free_tv_subst_var: "free_tv (S (v::nat)) ⊆ insert v (cod S)"
  apply (cases "v:dom S")
  apply (fastforce simp add: cod_def)
  apply (fastforce simp add: dom_def)
  done

lemma free_tv_app_subst_te: "free_tv ($ S (t::typ)) ⊆ cod S ∪ free_tv t"
  proof (induct t)
    case (TVar n) then show ?case by (simp add: free_tv_subst_var)
  next
    case (Fun t1 t2) then show ?case by fastforce
  qed

lemma free_tv_app_subst_type_scheme:
  "free_tv ($ S (sch::type_scheme)) ⊆ cod S ∪ free_tv sch"
  proof (induct sch)
    case (FVar n)
    then show ?case by (simp add: free_tv_subst_var)
  next
    case (BVar n)
    then show ?case by simp
  next
    case (SFun t1 t2)
    then show ?case by fastforce
  qed

lemma free_tv_app_subst_scheme_list: "free_tv ($ S (A::type_scheme list)) ⊆ cod S ∪
free_tv A"
  proof (induct A)
    case Nil then show ?case by simp
  next
    case (Cons a al)
    with free_tv_app_subst_type_scheme
    show ?case by fastforce
  qed

lemma free_tv_comp_subst:
  "free_tv (λu::nat. $ s1 (s2 u) :: typ) ⊆
  free_tv s1 ∪ free_tv s2"
  unfolding free_tv_subst dom_def
  by (force simp add: cod_def dom_def dest!:free_tv_app_subst_te [THEN subsetD])

```

```

lemma free_tv_o_subst:
  "free_tv ($ S1 o S2)  $\subseteq$  free_tv S1  $\cup$  free_tv (S2 :: nat => typ)"
unfolding o_def by (rule free_tv_comp_subst)

lemma free_tv_of_substitutions_extend_to_types:
  "n : free_tv t  $\implies$  free_tv (S n)  $\subseteq$  free_tv ($ S t::typ)"
by (induct t) auto

lemma free_tv_of_substitutions_extend_to_schemes:
  "n : free_tv sch  $\implies$  free_tv (S n)  $\subseteq$  free_tv ($ S sch::type_scheme)"
by (induct sch) auto

lemma free_tv_of_substitutions_extend_to_scheme_lists [simp]:
  "n : free_tv A  $\implies$  free_tv (S n)  $\subseteq$  free_tv ($ S A::type_scheme list)"
by (induct A) (auto dest: free_tv_of_substitutions_extend_to_schemes)

lemma free_tv_ML_scheme:
  fixes sch :: type_scheme
  shows "(n : free_tv sch) = (n: set (free_tv_ML sch))"
by (induct sch) simp_all

lemma free_tv_ML_scheme_list:
  fixes A :: "type_scheme list"
  shows "(n : free_tv A) = (n: set (free_tv_ML A))"
by (induct_tac A) (simp_all add: free_tv_ML_scheme)

— lemmata for bound_tv

lemma bound_tv_mk_scheme [simp]: "bound_tv (mk_scheme t) = {}"
by (induct t) simp_all

lemma bound_tv_subst_scheme [simp]:
  fixes sch :: type_scheme
  shows "bound_tv ($ S sch) = bound_tv sch"
by (induct sch) simp_all

lemma bound_tv_subst_scheme_list [simp]:
  fixes A :: "type_scheme list"
  shows "bound_tv ($ S A) = bound_tv A"
by (induct A) simp_all

— Lemmata for new_tv

lemma new_tv_subst:
  "new_tv n S = (( $\forall m. n \leq m \implies (S m = TVar m)$ )  $\wedge$ 
    ( $\forall l. 1 < n \implies new_tv n (S l)$  ))"
```

```

unfolding new_tv_def free_tv_subst cod_def dom_def
apply rule
  apply force
by simp (meson not_le)

lemma new_tv_list: "new_tv n x = (∀y∈set x. new_tv n y)"
by (induction x) simp_all

— substitution affects only variables occurring freely
lemma subst_te_new_tv:
  "new_tv n (t::typ) ⇒ $(λx. if x=n then t' else S x) t = $S t"
by (induct t) simp_all

lemma subst_te_new_type_scheme:
  "new_tv n (sch::type_scheme) ⇒ $(λx. if x=n then sch' else S x) sch = $S sch"
by (induct sch) simp_all

lemma subst_tel_new_scheme_list [simp]:
  "new_tv n (A::type_scheme list) ⇒ $(λx. if x=n then t else S x) A = $S A"
by (induct A) (simp_all add: subst_te_new_type_scheme)

— all greater variables are also new
lemma new_tv_le:
  "n ≤ m ⇒ new_tv n t ⇒ new_tv m t"
by (meson less_le_trans new_tv_def)

lemma new_tv_Suc[simp]: "new_tv n t ⇒ new_tv (Suc n) t"
by (rule lessI [THEN less_imp_le [THEN new_tv_le]])

— new_tv property remains if a substitution is applied
lemma new_tv_subst_var:
  "[| n < m; new_tv m (S::subst) |] ==> new_tv m (S n)"
by (simp add: new_tv_subst)

lemma new_tv_subst_te [simp]:
  "new_tv n S ⇒ new_tv n (t::typ) ⇒ new_tv n ($ S t)"
by (induction t) (auto simp add: new_tv_subst)

lemma new_tv_subst_scheme_list:
  "new_tv n S ⇒ new_tv n (A::type_scheme list) ⇒ new_tv n ($ S A)"
by (meson UnE codD free_tv_app_subst_scheme_list in_mono new_tv_def)

lemma new_tv_only_depends_on_free_tv_type_scheme:
  fixes sch :: type_scheme
  shows "free_tv sch = free_tv sch' ⇒ new_tv n sch ⇒ new_tv n sch'"
  unfolding new_tv_def by simp

lemma new_tv_only_depends_on_free_tv_scheme_list:

```

```

fixes A :: "type_scheme list"
shows "free_tv A = free_tv A'  $\implies$  new_tv n A  $\implies$  new_tv n A'"
unfolding new_tv_def by simp

lemma new_tv_nth_nat_A:
  "m < length A  $\implies$  new_tv n A  $\implies$  new_tv n (A!m)"
unfolding new_tv_def using free_tv_nth_A_impl_free_tv_A by blast

— composition of substitutions preserves new_tv proposition
lemma new_tv_subst_comp_1:
  "[| new_tv n (S::subst); new_tv n R |]  $\implies$  new_tv n (( $\$$  R)  $\circ$  S)"
by (simp add: new_tv_subst)

lemma new_tv_subst_comp_2 :
  "[| new_tv n (S::subst); new_tv n R |]  $\implies$  new_tv n ( $\lambda v.$   $\$$  R (S v))"
by (simp add: new_tv_subst)

— new type variables do not occur freely in a type structure
lemma new_tv_not_free_tv:
  "new_tv n A  $\implies$  n  $\notin$  free_tv A"
by (auto simp add: new_tv_def)

lemma fresh_variable_types: " $\exists n.$  new_tv n (t::typ)"
apply (unfold new_tv_def)
apply (induction t)
  apply auto[1]
by (metis less_trans linorder_cases new_tv_Fun new_tv_def)

lemma fresh_variable_type_schemes:
  " $\exists n.$  new_tv n (sch::type_scheme)"
apply (unfold new_tv_def)
apply (induct_tac sch)
  apply (auto)[1]
  apply simp
by (metis less_trans linorder_cases new_tv_Fun2 new_tv_def)

lemma fresh_variable_type_scheme_lists:
  " $\exists n.$  new_tv n (A::type_scheme list)"
apply (induction A)
  apply (simp)
by (metis fresh_variable_type_schemes le_cases new_tv_Cons new_tv_le)

lemma make_one_new_out_of_two:
  "[|  $\exists n1.$  (new_tv n1 x);  $\exists n2.$  (new_tv n2 y) |]  $\implies$   $\exists n.$  (new_tv n x)  $\wedge$  (new_tv n y)"
by (meson new_tv_le nle_le)

lemma ex_fresh_variable:

```

```

"^(A::type_scheme list) (A'::type_scheme list) (t::typ) (t'::typ).
  ∃n. (new_tv n A) ∧ (new_tv n A') ∧ (new_tv n t) ∧ (new_tv n t')"
by (meson fresh_variable_type_scheme_lists fresh_variable_types max.cobounded1 max.cobounded2
new_tv_le)

```

— mgu does not introduce new type variables

```

lemma mgu_new:
  "[|mgu t1 t2 = Some u; new_tv n t1; new_tv n t2|] ==> new_tv n u"
by (meson UnE mgu_free new_tv_def subsetD)

```

```

lemma length_app_subst_list [simp]:
  "^(A:: ('a::type_struct) list. length ($ S A) = length A"
unfolding app_subst_list by simp

```

```

lemma subst_TVar_scheme [simp]:
  fixes sch :: type_scheme
  shows "$ TVar sch = sch"
by (induct sch) simp_all

```

```

lemma subst_TVar_scheme_list [simp]:
  fixes A :: "type_scheme list"
  shows "$ TVar A = A"
by (induct A) (simp_all add: app_subst_list)

```

— application of `id_subst` does not change type expression

```

lemma app_subst_id_te [simp]: "$ id_subst = (λt::typ. t)"
by (metis id_subst_def mk_scheme_injective subst_TVar_scheme subst_mk_scheme)

```

```

lemma app_subst_id_type_scheme [simp]:
  "$ id_subst = (λsch::type_scheme. sch)"
using id_subst_def subst_TVar_scheme by auto

```

— application of `id_subst` does not change list of type expressions

```

lemma app_subst_id_tel [simp]:
  "$ id_subst = (λA::type_scheme list. A)"
using id_subst_def subst_TVar_scheme_list by auto

```

— composition of substitutions

```

lemma o_id_subst [simp]: "$S o id_subst = S"
unfolding id_subst_def o_def by simp

```

```

lemma subst_comp_te: "$ R ($ S t::typ) = $ (λx. $ R (S x) ) t"
by (induct t) simp_all

```

```

lemma subst_comp_type_scheme:

```

```
"$ R ($ S sch::type_scheme) = $ (\lambda x. $ R (S x) ) sch"
by (induct sch) simp_all
```

```
lemma subst_comp_scheme_list:
  "$ R ($ S A::type_scheme list) = $ (\lambda x. $ R (S x)) A"
unfolding app_subst_list
proof (induct A)
  case Nil thus ?case by simp
next
  case Cons thus ?case by (simp add: subst_comp_type_scheme)
qed
```

```
lemma nth_subst:
  "n < length A  $\implies$  ($ S A)!n = $S (A!n)"
by (simp add: app_subst_list)
```

end

### 3 Instances of type schemes

```
theory Instance
imports Type
begin
```

```
primrec bound_typ_inst :: "[subst, type_scheme] => typ" where
  "bound_typ_inst S (FVar n) = (TVar n)"
| "bound_typ_inst S (BVar n) = (S n)"
| "bound_typ_inst S (sch1 ==> sch2) = ((bound_typ_inst S sch1) -> (bound_typ_inst S sch2))"
```

```
primrec bound_scheme_inst :: "[nat => type_scheme, type_scheme] => type_scheme" where
  "bound_scheme_inst S (FVar n) = (FVar n)"
| "bound_scheme_inst S (BVar n) = (S n)"
| "bound_scheme_inst S (sch1 ==> sch2) = ((bound_scheme_inst S sch1) ==> (bound_scheme_inst S sch2))"
```

```
definition is_bound_typ_instance :: "[typ, type_scheme] => bool" (infixr "<|" 70) where
  is_bound_typ_instance: "t <| sch = ( $\exists$  S. t = (bound_typ_inst S sch))"
```

```
instantiation type_scheme :: ord
begin
```

```
definition
  le_type_scheme_def: "sch'  $\leq$  (sch::type_scheme)  $\iff$  ( $\forall$  t. t <| sch'  $\implies$  t <| sch)"
```

```
definition
  "(sch' < (sch :: type_scheme))  $\iff$  sch'  $\leq$  sch  $\wedge$  sch'  $\neq$  sch"
```

```
instance ..
```

end

```
primrec subst_to_scheme :: "[nat => type_scheme, typ] => type_scheme" where
  "subst_to_scheme B (TVar n) = (B n)"
| "subst_to_scheme B (t1 -> t2) = ((subst_to_scheme B t1) ==> (subst_to_scheme B t2))"
```

```
instantiation list :: (ord) ord
begin
```

definition

```
le_env_def: "A ≤ B ↔ length B = length A ∧ (∀i. i < length A → A!i ≤ B!i)"
```

definition

```
"(A < (B :: 'a list)) ↔ A ≤ B ∧ A ≠ B"
```

instance ..

end

lemmas for instantiation

```
lemma bound_typ_inst_mk_scheme [simp]: "bound_typ_inst S (mk_scheme t) = t"
  by (induct t) simp_all
```

```
lemma bound_typ_inst_composed_subst [simp]:
  "bound_typ_inst ($S ∘ R) ($S sch) = $S (bound_typ_inst R sch)"
  by (induct sch) simp_all
```

```
lemma bound_typ_inst_eq:
  "S = S' ⇒ sch = sch' ⇒ bound_typ_inst S sch = bound_typ_inst S' sch'"
  by simp
```

```
lemma bound_scheme_inst_mk_scheme [simp]:
  "bound_scheme_inst B (mk_scheme t) = mk_scheme t"
  by (induct t) simp_all
```

```
lemma substitution_lemma: "$S (bound_scheme_inst B sch) = (bound_scheme_inst ($S ∘ B)
($ S sch))"
  by (induct sch) simp_all
```

```
lemma bound_scheme_inst_type [rule_format]: "∀t. mk_scheme t = bound_scheme_inst B sch
→
```

```
  (∃S. ∀x∈bound_tv sch. B x = mk_scheme (S x))"
```

```
apply (induct_tac "sch")
apply (simp (no_asm))
apply safe
apply (rule exI)
apply (rule ballI)
apply (rule sym)
apply simp
```

```

apply (rename_tac type_scheme1 type_scheme2 t)
apply simp
apply (drule mk_scheme_Fun)
apply (erule exE)+
apply (erule conjE)
apply (drule sym)
apply (drule sym)
apply (drule mp, fast)+
apply safe
apply (rename_tac S1 S2)
apply (rule_tac x = "\lambda x. if x:bound_tv type_scheme1 then (S1 x) else (S2 x) " in exI)
apply auto
done

```

```

lemma subst_to_scheme_inverse:
  "new_tv n sch  $\implies$ 
  subst_to_scheme ( $\lambda k. \text{if } n \leq k \text{ then BVar } (k - n) \text{ else FVar } k$ )
  (bound_typ_inst ( $\lambda k. \text{TVar } (k + n)$ ) sch) = sch"
apply (induct sch)
  apply (simp add: not_le)
  apply (simp add: le_add2 diff_add_inverse2)
apply simp
done

```

```

lemma aux: "t = t'  $\implies$ 
  subst_to_scheme ( $\lambda k. \text{if } n \leq k \text{ then BVar } (k - n) \text{ else FVar } k$ ) t =
  subst_to_scheme ( $\lambda k. \text{if } n \leq k \text{ then BVar } (k - n) \text{ else FVar } k$ ) t'"
by blast

```

```

lemma aux2: "new_tv n sch  $\implies$ 
  subst_to_scheme ( $\lambda k. \text{if } n \leq k \text{ then BVar } (k - n) \text{ else FVar } k$ ) (bound_typ_inst S sch)
=
  bound_scheme_inst ((subst_to_scheme ( $\lambda k. \text{if } n \leq k \text{ then BVar } (k - n) \text{ else FVar } k$ ))
  o S) sch"
  by (induct sch) auto

```

```

lemma le_type_scheme_def2:
  fixes sch sch' :: type_scheme
  shows "(sch'  $\leq$  sch) = ( $\exists B. \text{sch}' = \text{bound\_scheme\_inst } B \text{ sch}$ )"
apply (unfold le_type_scheme_def is_bound_typ_instance)
apply (rule iffI)
apply (cut_tac sch = "sch" in fresh_variable_type_schemes)
apply (cut_tac sch = "sch'" in fresh_variable_type_schemes)
apply (drule make_one_new_out_of_two)
apply assumption
apply (erule_tac V = " $\exists n. \text{new\_tv } n \text{ sch}'$ " in thin_rl)
apply (erule exE)
apply (erule allE)
apply (drule mp)

```



```

apply (rule_tac x = " ( $\lambda k. TVar (k + n)$ )" in exI)
apply (rule refl)
apply (erule exE)
apply (erule conjE)+
apply (drule_tac n = "n" in aux)
apply (simp add: subst_to_scheme_inverse)
apply (rule_tac x = " (subst_to_scheme ( $\lambda k. \text{if } n \leq k \text{ then } BVar (k - n) \text{ else } FVar k$ ))
 $\circ S$ " in exI)
apply (simp (no_asm_simp) add: aux2)
apply safe
apply (rule_tac x = " $\lambda n. \text{bound\_typ\_inst } S (B n)$ " in exI)
apply (induct_tac "sch")
apply (simp (no_asm))
apply (simp (no_asm))
apply (simp (no_asm_simp))
done

```

```

lemma le_type_eq_is_bound_typ_instance [rule_format]: "(mk_scheme t)  $\leq$  sch = t <| sch"
apply (unfold is_bound_typ_instance)
apply (simp (no_asm) add: le_type_scheme_def2)
apply (rule iffI)
apply (erule exE)
apply (frule bound_scheme_inst_type)
apply (erule exE)
apply (rule exI)
apply (rule mk_scheme_injective)
apply simp
apply (rotate_tac 1)
apply (rule mp)
prefer 2 apply (assumption)
apply (induct_tac "sch")
apply (simp (no_asm))
apply simp
apply fast
apply (intro strip)
apply simp
apply (erule exE)
apply simp
apply (rule exI)
apply (induct_tac "sch")
apply (simp (no_asm))
apply (simp (no_asm))
apply simp
done

```

```

lemma le_env_Cons [iff]:
  "(sch # A  $\leq$  sch' # B) = (sch  $\leq$  (sch'::type_scheme)  $\wedge$  A  $\leq$  B)"
apply (unfold le_env_def)
apply (simp (no_asm))

```

```

apply (rule iffI)
  apply clarify
  apply (rule conjI)
    apply (erule_tac x = "0" in allE)
    apply simp
  apply (rule conjI, assumption)
  apply clarify
  apply (erule_tac x = "Suc i" in allE)
  apply simp
apply (rule conjI)
  apply fast
apply (rule allI)
apply (induct_tac "i")
apply simp_all
done

```

```

lemma is_bound_typ_instance_closed_subst: "t <| sch  $\implies$   $\$S$  t <|  $\$S$  sch"
apply (unfold is_bound_typ_instance)
apply (erule exE)
apply (rename_tac "SA")
apply simp
apply (rule_tac x = " $\$S \circ SA$ " in exI)
apply simp
done

```

```

lemma S_compatible_le_scheme:
  fixes sch sch' :: type_scheme
  shows "sch'  $\leq$  sch  $\implies$   $\$S$  sch'  $\leq$   $\$ S$  sch"
apply (simp add: le_type_scheme_def2)
apply (erule exE)
apply (simp add: substitution_lemma)
apply fast
done

```

```

lemma S_compatible_le_scheme_lists:
  fixes A A' :: "type_scheme list"
  shows "A'  $\leq$  A  $\implies$   $\$S$  A'  $\leq$   $\$ S$  A"
apply (unfold le_env_def app_subst_list)
apply (simp cong add: conj_cong)
apply (fast intro!: S_compatible_le_scheme)
done

```

```

lemma bound_typ_instance_trans: "[| t <| sch; sch  $\leq$  sch' |]  $\implies$  t <| sch'"
  unfolding le_type_scheme_def by blast

```

```

lemma le_type_scheme_refl [iff]: "sch  $\leq$  (sch::type_scheme)"
  unfolding le_type_scheme_def by blast

```

```

lemma le_env_refl [iff]: "A  $\leq$  (A::type_scheme list)"

```

```

unfolding le_env_def by blast

lemma bound_typ_instance_BVar [iff]: "sch ≤ BVar n"
apply (unfold le_type_scheme_def is_bound_typ_instance)
apply (intro strip)
apply (rule_tac x = "λa. t" in exI)
apply simp
done

lemma le_FVar [simp]: "(sch ≤ FVar n) = (sch = FVar n)"
apply (unfold le_type_scheme_def is_bound_typ_instance)
apply (induct_tac "sch")
apply auto
done

lemma not_FVar_le_Fun [iff]: "¬(FVar n ≤ sch1 ==> sch2)"
  unfolding le_type_scheme_def is_bound_typ_instance by simp

lemma not_BVar_le_Fun [iff]: "¬(BVar n ≤ sch1 ==> sch2)"
apply (unfold le_type_scheme_def is_bound_typ_instance)
apply simp
apply (rule_tac x = "TVar n" in exI)
apply fastforce
done

lemma Fun_le_FunD:
  "(sch1 ==> sch2 ≤ sch1' ==> sch2') ==> sch1 ≤ sch1' ∧ sch2 ≤ sch2'"
  unfolding le_type_scheme_def is_bound_typ_instance by fastforce

lemma scheme_le_Fun: "(sch' ≤ sch1 ==> sch2) ==> ∃sch'1 sch'2. sch' = sch'1 ==> sch'2"
  by (induct sch') auto

lemma le_type_scheme_free_tv [rule_format]:
  "∀sch'::type_scheme. sch ≤ sch' ==> free_tv sch' ≤ free_tv sch"
apply (induct_tac "sch")
  apply (rule allI)
  apply (induct_tac "sch'")
    apply (simp (no_asm))
    apply (simp (no_asm))
    apply (simp (no_asm))
  apply (rule allI)
  apply (induct_tac "sch'")
    apply (simp (no_asm))
    apply (simp (no_asm))
    apply (simp (no_asm))
  apply (rule allI)
  apply (induct_tac "sch'")
    apply (simp (no_asm))
    apply (simp (no_asm))

```

```

apply simp
apply (intro strip)
apply (drule Fun_le_FunD)
apply fast
done

lemma le_env_free_tv [rule_format]:
  "∀A::type_scheme list. A ≤ B → free_tv B ≤ free_tv A"
apply (induct_tac "B")
  apply (simp (no_asm))
apply (rule allI)
apply (induct_tac "A")
  apply (simp (no_asm) add: le_env_def)
  apply (simp (no_asm))
apply (fast dest: le_type_scheme_free_tv)
done

end

```

## 4 Generalizing type schemes with respect to a context

```

theory Generalize
imports Instance
begin

— gen: binding (generalising) the variables which are not free in the context

type_synonym ctxt = "type_scheme list"

primrec gen :: "[ctxt, typ] => type_scheme" where
  "gen A (TVar n) = (if (n:(free_tv A)) then (FVar n) else (BVar n))"
| "gen A (t1 -> t2) = (gen A t1) ==> (gen A t2)"

— executable version of gen: implementation with free_tv_ML

primrec gen_ML_aux :: "[nat list, typ] => type_scheme" where
  "gen_ML_aux A (TVar n) = (if (n: set A) then (FVar n) else (BVar n))"
| "gen_ML_aux A (t1 -> t2) = (gen_ML_aux A t1) ==> (gen_ML_aux A t2)"

definition gen_ML :: "[ctxt, typ] => type_scheme" where
  gen_ML_def: "gen_ML A t = gen_ML_aux (free_tv_ML A) t"

declare equalityE [elim!]

lemma gen_eq_on_free_tv:
  "free_tv A = free_tv B ==> gen A t = gen B t"
  by (induct t) simp_all

lemma gen_without_effect [simp]:

```

```

    "(free_tv t) ⊆ (free_tv sch) ⇒ gen sch t = (mk_scheme t)"
  by (induct t) auto

lemma free_tv_gen [simp]:
  "free_tv (gen ($ S A) t) = free_tv t Int free_tv ($ S A)"
  by (induct t) auto

lemma free_tv_gen_cons [simp]:
  "free_tv (gen ($ S A) t # $ S A) = free_tv ($ S A)"
  by fastforce

lemma bound_tv_gen [simp]:
  "bound_tv (gen A t) = free_tv t - free_tv A"
  by (induction t) auto

lemma new_tv_compatible_gen: "new_tv n t ⇒ new_tv n (gen A t)"
  by (induction t) auto

lemma gen_eq_gen_ML: "gen A t = gen_ML A t"
  apply (unfold gen_ML_def)
  apply (induct t)
  apply (simp add: free_tv_ML_scheme_list)
  apply (simp add: free_tv_ML_scheme_list)
  done

lemma gen_subst_commutates [rule_format]:
  "(free_tv S) Int ((free_tv t) - (free_tv A)) = {}
   → gen ($ S A) ($ S t) = $ S (gen A t)"
  apply (induct t)
  apply (intro strip)
  apply (rename_tac nat)
  apply (case_tac "nat : (free_tv A) ")
  apply (simp (no_asm_simp))
  apply simp
  apply (subgoal_tac "nat ∉ free_tv S")
  prefer 2 apply (fast)
  apply (simp add: free_tv_subst_dom_def)
  apply (cut_tac free_tv_app_subst_scheme_list)
  apply fast
  apply simp
  apply blast
  done

lemma bound_typ_inst_gen [simp]:
  "free_tv(t::typ) ⊆ free_tv(A) ⇒ bound_typ_inst S (gen A t) = t"
  by (induct t) simp_all

lemma gen_bound_typ_instance:
  "gen ($ S A) ($ S t) ≤ $ S (gen A t)"

```

```

apply (unfold le_type_scheme_def is_bound_typ_instance)
apply safe
apply (rename_tac "R")
apply (rule_tac x = " ( $\lambda a. \text{bound\_typ\_inst } R \text{ (gen } (\$S \ A) \ (S \ a))$ ) " in exI)
apply (induct_tac "t")
  apply simp
apply simp
done

lemma free_tv_subset_gen_le:
  "free_tv B  $\subseteq$  free_tv A  $\implies$  gen A t  $\leq$  gen B t"
apply (unfold le_type_scheme_def is_bound_typ_instance)
apply safe
apply (rename_tac "S")
apply (rule_tac x = " $\lambda b. \text{if } b:\text{free\_tv } A \text{ then } TVar \ b \text{ else } S \ b$ " in exI)
apply (induct_tac "t")
  apply fastforce
apply simp
done

lemma gen_t_le_gen_alpha_t [rule_format, simp]:
  "new_tv n A  $\longrightarrow$ 
  gen A t  $\leq$  gen A ( $\$ (\lambda x. TVar \ (\text{if } x \in \text{free\_tv } A \text{ then } x \text{ else } n + x)) \ t$ )"
apply (unfold le_type_scheme_def is_bound_typ_instance)
apply (intro strip)
apply (erule exE)
apply (hypsubst)
apply (rule_tac x = " ( $\lambda x. S \ (\text{if } n \leq x \text{ then } x - n \text{ else } x)$ )" in exI)
apply (induct t)
apply (simp (no_asm))
apply (rename_tac nat S)
apply (case_tac "nat  $\in$  free_tv A")
apply (simp (no_asm_simp))
apply (simp (no_asm_simp))
apply (subgoal_tac "n  $\leq$  n + nat")
apply (frule_tac t = "A" in new_tv_le)
apply assumption
apply (drule new_tv_not_free_tv)
apply (drule new_tv_not_free_tv)
apply (simp add: diff_add_inverse)
apply (simp add: le_add1)
apply simp
done

```

end

## 5 MiniML with type inference rules

theory MiniML

```

imports Generalize
begin

— expressions
datatype
  expr = Var nat | Abs expr | App expr expr | LET expr expr

— type inference rules
inductive
  has_type :: "[cxtxt, expr, typ] => bool"
            ("((_) ⊢/ (_) :: (_))" [60,0,60] 60)
where
  VarI: "[| n < length A; t <| A!n |] ==> A ⊢ Var n :: t"
| AbsI: "[| (mk_scheme t1)#A ⊢ e :: t2 |] ==> A ⊢ Abs e :: t1 -> t2"
| AppI: "[| A ⊢ e1 :: t2 -> t1; A ⊢ e2 :: t2 |]
        ==> A ⊢ App e1 e2 :: t1"
| LETI: "[| A ⊢ e1 :: t1; (gen A t1)#A ⊢ e2 :: t |] ==> A ⊢ LET e1 e2 :: t"

declare has_type.intros [simp]
declare Un_upper1 [simp] Un_upper2 [simp]
declare is_bound_typ_instance_closed_subst [simp]

lemma s'_t_equals_s_t[simp]:
  "∧t::typ. $(λn. if n : (free_tv A) Un (free_tv t) then (S n) else (TVar n)) t = $S
  t"
apply (rule typ_substitutions_only_on_free_variables)
apply (simp add: Ball_def)
done

lemma s'_a_equals_s_a [simp]:
  "∧A::type_scheme list. $(λn. if n : (free_tv A) Un (free_tv t) then (S n) else (TVar
  n)) A = $S A"
apply (rule scheme_list_substitutions_only_on_free_variables)
apply (simp add: Ball_def)
done

lemma replace_s_by_s':
  "$$(λn. if n : (free_tv A) Un (free_tv t) then S n else TVar n) A ⊢
  e :: $(λn. if n : (free_tv A) Un (free_tv t) then S n else TVar n) t
  ⇒ $S A ⊢ e :: $S t"
apply (rule_tac P = "λA. A ⊢ e :: $S t" in ssubst)
apply (rule s'_a_equals_s_a [symmetric])
apply (rule_tac P = "λt. $(λn. if n : free_tv A Un free_tv (t2 S) then S n else TVar
  n) A ⊢ e :: t" for t2 in ssubst)
apply (rule s'_t_equals_s_t [symmetric])
apply simp
done

lemma alpha_A':

```

```

"∧A::type_scheme list. $ (λx. TVar (if x : free_tv A then x else n + x)) A = $ id_subst
A"
apply (rule scheme_list_substitutions_only_on_free_variables)
apply (simp add: id_subst_def)
done

```

```

lemma alpha_A:
"∧A::type_scheme list. $ (λx. TVar (if x : free_tv A then x else n + x)) A = A"
apply (rule alpha_A' [THEN ssubst])
apply simp
done

```

```

lemma S_o_alpha_typ:
"$ (S o alpha) (t::typ) = $ S ($ (λx. TVar (alpha x)) t)"
apply (induct_tac "t")
apply (simp_all)
done

```

```

lemma S_o_alpha_typ':
"$ (λu. (S o alpha) u) (t::typ) = $ S ($ (λx. TVar (alpha x)) t)"
apply (induct_tac "t")
apply (simp_all)
done

```

```

lemma S_o_alpha_type_scheme:
"$ (S o alpha) (sch::type_scheme) = $ S ($ (λx. TVar (alpha x)) sch)"
apply (induct_tac "sch")
apply (simp_all)
done

```

```

lemma S_o_alpha_type_scheme_list:
"$ (S o alpha) (A::type_scheme list) = $ S ($ (λx. TVar (alpha x)) A)"
apply (induct_tac "A")
apply (simp_all)
apply (rule S_o_alpha_type_scheme [unfolded o_def])
done

```

```

lemma S'_A_eq_S'_alpha_A: "∧A::type_scheme list.
$ (λn. if n : free_tv A Un free_tv t then S n else TVar n) A =
$ ((λx. if x : free_tv A Un free_tv t then S x else TVar x) o
(λx. if x : free_tv A then x else n + x)) A"
apply (subst S_o_alpha_type_scheme_list)
apply (subst alpha_A)
apply (rule refl)
done

```

```

lemma dom_S':
"dom (λn. if n : free_tv A Un free_tv t then S n else TVar n) ⊆
free_tv A Un free_tv t"

```



```

apply (unfold free_tv_subst dom_def)
apply (simp (no_asm))
apply fast
done

```

```

lemma cod_S':

```

```

  " $\bigwedge(A::\text{type\_scheme list}) (t::\text{typ}).$ 
  cod  $(\lambda n. \text{if } n : \text{free\_tv } A \text{ Un free\_tv } t \text{ then } S \ n \text{ else } \text{TVar } n) \subseteq$ 
  free_tv  $(\$ S \ A) \text{ Un free\_tv } (\$ S \ t)''$ 
apply (unfold free_tv_subst cod_def subset_eq)
apply (rule ballI)
apply (erule UN_E)
apply (drule dom_S' [THEN subsetD])
apply simp
apply (fast dest: free_tv_of_substitutions_extend_to_scheme_lists intro: free_tv_of_substitutions_e
[THEN subsetD])
done

```

```

lemma free_tv_S':

```

```

  " $\bigwedge(A::\text{type\_scheme list}) (t::\text{typ}).$ 
  free_tv  $(\lambda n. \text{if } n : \text{free\_tv } A \text{ Un free\_tv } t \text{ then } S \ n \text{ else } \text{TVar } n) \subseteq$ 
  free_tv  $A \text{ Un free\_tv } (\$ S \ A) \text{ Un free\_tv } t \text{ Un free\_tv } (\$ S \ t)''$ 
apply (unfold free_tv_subst)
apply (fast dest: dom_S' [THEN subsetD] cod_S' [THEN subsetD])
done

```

```

lemma free_tv_alpha: " $\bigwedge t1::\text{typ}.$ 

```

```

  (free_tv  $(\$ (\lambda x. \text{TVar } (\text{if } x : \text{free\_tv } A \text{ then } x \text{ else } n + x)) t1) - \text{free\_tv } A) \subseteq$ 
   $\{x. \exists y. x = n + y\}''$ 
apply (induct_tac "t1")
apply (simp (no_asm))
apply fast
apply (simp (no_asm))
apply fast
done

```

```

lemma new_tv_Int_free_tv_empty_type: " $\bigwedge t::\text{typ}.$  new_tv n t  $\implies \{x. \exists y. x = n + y\} \cap$ 
free_tv t =  $\{\}$ "

```

```

apply safe
apply (cut_tac le_add1)
apply (drule new_tv_le)
apply assumption
apply (rotate_tac 1)
apply (drule new_tv_not_free_tv)
apply fast
done

```

```

lemma new_tv_Int_free_tv_empty_scheme:

```

```

  " $\bigwedge \text{sch}::\text{type\_scheme}.$  new_tv n sch  $\implies \{x. \exists y. x = n + y\} \cap \text{free\_tv sch} = \{\}$ "

```

```

apply safe
apply (cut_tac le_add1)
apply (drule new_tv_le)
apply assumption
apply (rotate_tac 1)
apply (drule new_tv_not_free_tv)
apply fast
done

```

```

lemma new_tv_Int_free_tv_empty_scheme_list:
  " $\forall A::\text{type\_scheme list. new\_tv } n \ A \longrightarrow \{x. \exists y. x = n + y\} \cap \text{free\_tv } A = \{\}$ "
apply (rule allI)
apply (induct_tac "A")
apply (simp (no_asm))
apply (simp (no_asm))
apply (fast dest: new_tv_Int_free_tv_empty_scheme)
done

```

```

lemma gen_t_le_gen_alpha_t [rule_format (no_asm)]:
  " $\text{new\_tv } n \ A \longrightarrow \text{gen } A \ t \leq \text{gen } A \ (\lambda x. \text{TVar } (\text{if } x : \text{free\_tv } A \text{ then } x \text{ else } n + x))$ 
  t)"
apply (unfold le_type_scheme_def is_bound_typ_instance)
apply (intro strip)
apply (erule exE)
apply (hypsubst)
apply (rule_tac x = " $\lambda x. S \ (\text{if } n \leq x \text{ then } x - n \text{ else } x)$ " in exI)
apply (induct_tac t)
apply (simp (no_asm))
apply (rename_tac nat)
apply (case_tac "nat : free_tv A")
apply (simp (no_asm_simp))
apply (subgoal_tac "n ≤ n + nat")
apply (drule new_tv_le)
apply assumption
apply (drule new_tv_not_free_tv)
apply (drule new_tv_not_free_tv)
apply (simp (no_asm_simp) add: diff_add_inverse)
apply (simp (no_asm))
apply (simp (no_asm_simp))
done

```

```

declare has_type.intros [intro!]

```

```

lemma has_type_le_env [rule_format (no_asm)]: " $A \vdash e::t \implies \forall B. A \leq B \longrightarrow B \vdash e::t$ "
apply (erule has_type.induct)
  apply (simp (no_asm) add: le_env_def)
  apply (fastforce elim: bound_typ_instance_trans)
  apply simp
  apply fast

```

```

apply (slow elim: le_env_free_tv [THEN free_tv_subset_gen_le])
done

— has_type is closed w.r.t. substitution
lemma has_type_cl_sub: "A ⊢ e :: t ⇒ ∀S. $S A ⊢ e :: $S t"
apply (erule has_type.induct)

  apply (rule allI)
  apply (rule has_type.VarI)
  apply (simp add: app_subst_list)
  apply (simp (no_asm_simp) add: app_subst_list)

  apply (rule allI)
  apply (simp (no_asm))
  apply (rule has_type.AbsI)
  apply simp

  apply (rule allI)
  apply (rule has_type.AppI)
  apply simp
  apply (erule spec)
  apply (erule spec)

  apply (rule allI)
  apply (rule replace_s_by_s')
  apply (cut_tac A = "$ S A" and A' = "A" and t = "t" and t' = "$ S t" in ex_fresh_variable)
  apply (erule exE)
  apply (erule conjE)+
  apply (rule_tac ?t1.0 = "$ ((λx. if x : free_tv A Un free_tv t then S x else TVar x) ◦
(λx. if x : free_tv A then x else n + x)) t1" in has_type.LETI)
  apply (drule_tac x = " (λx. if x : free_tv A Un free_tv t then S x else TVar x) ◦ (λx.
if x : free_tv A then x else n + x) " in spec)
  apply (subst S'_A_eq_S'_alpha_A)
  apply assumption
  apply (subst S_o_alpha_typ)
  apply (subst gen_subst_commutes)
  apply (rule subset_antisym)
  apply (rule subsetI)
  apply (erule IntE)
  apply (drule free_tv_S' [THEN subsetD])
  apply (drule free_tv_alpha [THEN subsetD])
  apply (simp)
  apply (erule exE)
  apply (hypsubst)
  apply (subgoal_tac "new_tv (n + y) ($ S A) ")
  apply (subgoal_tac "new_tv (n + y) ($ S t) ")
  apply (subgoal_tac "new_tv (n + y) A")
  apply (subgoal_tac "new_tv (n + y) t")
  apply (drule new_tv_not_free_tv)+

```

```

    apply fast
    apply (rule new_tv_le) prefer 2 apply assumption apply simp
    apply (rule new_tv_le) prefer 2 apply assumption apply simp
    apply (rule new_tv_le) prefer 2 apply assumption apply simp
    apply (rule new_tv_le) prefer 2 apply assumption apply simp
  apply fast
apply (rule has_type_le_env)
  apply (drule spec)
  apply (drule spec)
  apply assumption
apply (rule app_subst_Cons [THEN subst])
apply (rule S_compatible_le_scheme_lists)
apply (simp (no_asm_simp))
done

```

end

## 6 Correctness and completeness of type inference algorithm W

```

theory W
imports MiniML
begin

type_synonym result_W = "(subst * typ * nat) option"

— type inference algorithm W
fun W :: "[expr, ctxt, nat] => result_W" where
  "W (Var i) A n =
    (if i < length A then Some( id_subst,
      bound_typ_inst (λb. TVar(b+n)) (A!i),
      n + (min_new_bound_tv (A!i)) )
    else None)"

| "W (Abs e) A n = ( (S,t,m) := W e ((FVar n)#A) (Suc n);
  Some( S, (S n) -> t, m ) )"

| "W (App e1 e2) A n = ( (S1,t1,m1) := W e1 A n;
  (S2,t2,m2) := W e2 ($S1 A) m1;
  U := mgu ($S2 t1) (t2 -> (TVar m2));
  Some( $U o $S2 o S1, U m2, Suc m2 ) )"

| "W (LET e1 e2) A n = ( (S1,t1,m1) := W e1 A n;
  (S2,t2,m2) := W e2 ((gen ($S1 A) t1)#($S1 A)) m1;
  Some( $S2 o S1, t2, m2 ) )"

```

```
declare Suc_le_lessD [simp]
```

```
inductive_cases has_type_casesE:
```

```
"A ⊢ Var n :: t"
"A ⊢ Abs e :: t"
"A ⊢ App e1 e2 :: t"
"A ⊢ LET e1 e2 :: t"
```

— the resulting type variable is always greater or equal than the given one

```
lemma W_var_ge:
```

```
"W e A n = Some (S,t,m) ⇒ n ≤ m"
```

```
proof (induction e arbitrary: A n S t m)
```

```
  case Var thus ?case by (auto split: if_splits)
```

```
next
```

```
  case Abs thus ?case by (fastforce split: split_option_bind_asm)
```

```
next
```

```
  case App thus ?case by (fastforce split: split_option_bind_asm)
```

```
next
```

```
  case LET thus ?case by (fastforce split: split_option_bind_asm)
```

```
qed
```

```
declare W_var_ge [simp]
```

```
lemma W_var_geD:
```

```
"Some (S,t,m) = W e A n ⇒ n ≤ m"
```

```
by (metis W_var_ge)
```

```
lemma new_tv_compatible_W:
```

```
"new_tv n A ⇒ Some (S,t,m) = W e A n ⇒ new_tv m A"
```

```
by (metis W_var_ge new_tv_le)
```

```
lemma new_tv_bound_typ_inst_sch:
```

```
"new_tv n sch ⇒ new_tv (n + (min_new_bound_tv sch)) (bound_typ_inst (λb. TVar (b + n)) sch)"
```

```
proof (induction sch)
```

```
  case FVar thus ?case by simp
```

```
next
```

```
  case BVar thus ?case by simp
```

```
next
```

```
  case SFun thus ?case by (auto simp add: max_def nle_le dest: new_tv_le add_left_mono)
```

```
qed
```

— resulting type variable is new

```
lemma new_tv_W [rule_format]:
```

```
"∀n A S t m. new_tv n A → W e A n = Some (S,t,m) →
  new_tv m S ∧ new_tv m t"
```

```
proof (induction e)
```

```

    case Var thus ?case
      by (auto simp add: new_tv_bound_typ_inst_sch dest: new_tv_nth_nat_A)
next
  case Abs thus ?case
    apply (simp add: new_tv_subst split: split_option_bind)
    by (metis lessI new_tv_Cons new_tv_FVar new_tv_Suc new_tv_compatible_W )
next
  case App thus ?case
    apply (simp split: split_option_bind)
    by (smt (verit, ccfv_threshold) W_var_geD fun.map_comp lessI mgu_new new_tv_Fun new_tv_Suc
new_tv_le new_tv_subst new_tv_subst_comp_1 new_tv_subst_scheme_list new_tv_subst_te)
next
  case LET thus ?case
    apply (simp split: split_option_bind)
    by (metis W_var_ge new_tv_Cons new_tv_compatible_gen new_tv_le new_tv_subst_comp_1
new_tv_subst_scheme_list)
qed

```

lemma free\_tv\_bound\_typ\_inst1:

```

"v ∉ free_tv sch ⇒ v ∈ free_tv (bound_typ_inst (TVar ∘ S) sch) ⇒ ∃x. v = S x"
by (induction sch) auto

```

lemma free\_tv\_W [rule\_format]:

```

"∀n A S t m v. W e A n = Some (S,t,m) →
(v ∈ free_tv S ∨ v ∈ free_tv t) → v < n → v ∈ free_tv A"

```

proof (induct e)

case (Var n) then show ?case

```

  apply (simp (no_asm) add: free_tv_subst split: split_option_bind)
  apply (intro strip)
  apply (case_tac "v : free_tv (A!n)")
  apply (simp add: free_tv_nth_A_impl_free_tv_A)
  apply (drule free_tv_bound_typ_inst1)
  apply (simp (no_asm) add: o_def)
  apply (erule exE)
  apply simp
done

```

case Abs then show ?case

```

  apply (simp add: free_tv_subst split: split_option_bind del: all_simps)
  apply (intro strip)
  apply (rename_tac S t n1 v)
  apply (erule_tac x = "Suc n" in allE)
  apply (erule_tac x = "FVar n # A" in allE)
  apply (erule_tac x = "S" in allE)
  apply (erule_tac x = "t" in allE)
  apply (erule_tac x = "n1" in allE)
  apply (erule_tac x = "v" in allE)
  apply (bestsimp intro: cod_app_subst simp add: less_Suc_eq)
done

```

case App then show ?case

```

apply (simp (no_asm) split: split_option_bind del: all_simps)
apply (intro strip)
apply (rename_tac S t n1 S1 t1 n2 S2 v)
apply (erule_tac x = "n" in allE)
apply (erule_tac x = "A" in allE)
apply (erule_tac x = "S" in allE)
apply (erule_tac x = "t" in allE)
apply (erule_tac x = "n1" in allE)
apply (erule_tac x = "n1" in allE)
apply (erule_tac x = "v" in allE)

apply (erule_tac x = "$ S A" in allE)
apply (erule_tac x = "S1" in allE)
apply (erule_tac x = "t1" in allE)
apply (erule_tac x = "n2" in allE)
apply (erule_tac x = "v" in allE)
apply (intro conjI impI | elim conjE)+
apply (simp add: rotate_Some o_def)
apply (drule W_var_geD)
apply (drule W_var_geD)
apply ( (frule less_le_trans) , (assumption))
apply clarsimp
apply (drule free_tv_comp_subst [THEN subsetD])
apply (drule sym [THEN mgu_free])
apply clarsimp
apply (fastforce dest: free_tv_comp_subst [THEN subsetD] sym [THEN mgu_free] codD
free_tv_app_subst_te [THEN subsetD] free_tv_app_subst_scheme_list [THEN subsetD] less_le_trans
less_not_refl2 subsetD)
  apply simp
  apply (drule sym [THEN W_var_geD])
  apply (drule sym [THEN W_var_geD])
  apply ( (frule less_le_trans) , (assumption))
  apply clarsimp
  apply (drule mgu_free)
  apply (fastforce dest: mgu_free codD free_tv_subst_var [THEN subsetD] free_tv_app_subst_te
[THEN subsetD] free_tv_app_subst_scheme_list [THEN subsetD] less_le_trans subsetD)
done
next
case LET then show ?case
  apply (simp (no_asm) split: split_option_bind del: all_simps)
  apply (intro strip)
  apply (rename_tac S1 t1 n2 S2 t2 n3 v)
  apply (erule_tac x = "n" in allE)
  apply (erule_tac x = "A" in allE)
  apply (erule_tac x = "S1" in allE)
  apply (erule_tac x = "t1" in allE)
  apply (rotate_tac -1)
  apply (erule_tac x = "n2" in allE)
  apply (rotate_tac -1)

```

```

    apply (erule_tac x = "v" in allE)
    apply (erule (1) notE impE)
    apply (erule allE,erule allE,erule allE,erule allE,erule allE,erule_tac x = "v" in
allE)
    apply (erule (1) notE impE)
    apply simp
    apply (rule conjI)
    apply (fast dest!: codD free_tv_app_subst_scheme_list [THEN subsetD] free_tv_o_subst
[THEN subsetD] W_var_ge dest: less_le_trans)
    apply (fast dest!: codD free_tv_app_subst_scheme_list [THEN subsetD] W_var_ge dest:
less_le_trans)
    done
qed

```

```

lemma weaken_A_Int_B_eq_empty: " $(\forall x. x \in A \longrightarrow x \notin B) \implies A \cap B = \{\}$ "
apply fast
done

```

```

lemma weaken_not_elem_A_minus_B: " $x \notin A \vee x \in B \implies x \notin A - B$ "
apply fast
done

```

— correctness of  $W$  with respect to  $has\_type$

```

lemma W_correct_lemma [rule_format]: " $\forall A S t m n . new\_tv\ n\ A \longrightarrow \text{Some } (S,t,m) = W\ e\ A\ n \longrightarrow \$S\ A \vdash e :: t$ "
proof (induct "e")
case (Var n) thus ?case
apply simp
apply (intro strip)
apply (rule has_type.VarI)
apply (simp (no_asm))
apply (simp (no_asm) add: is_bound_typ_instance)
apply (rule exI)
apply (rule refl)
done
case (Abs e) thus ?case
apply (simp add: app_subst_list_split: split_option_bind)
apply (intro strip)
apply (erule_tac x = " (mk_scheme (TVar n)) # A" in allE)
apply simp
apply (rule has_type.AbsI)
apply (drule le_refl [THEN le_SucI, THEN new_tv_le])
apply (drule sym)
apply (erule allE)+
apply (erule impE)
apply (erule_tac [2] notE impE, tactic "assume_tac @{context} 2")
apply (simp (no_asm_simp))
by assumption
case (App e1 e2) thus ?case

```



```

apply (simp (no_asm) split: split_option_bind)
apply (intro strip)
apply (rename_tac S1 t1 n1 S2 t2 n2 S3)
apply (rule_tac ?t2.0 = "$ S3 t2" in has_type.AppI)
apply (rule_tac S1 = "S3" in app_subst_TVar [THEN subst])
apply (rule app_subst_Fun [THEN subst])
apply (rule_tac t = "$S3 (t2 -> (TVar n2))" and s = "$S3 ($S2 t1) " in subst)
apply (simp add: mgu_eq)
apply (simp only: subst_comp_scheme_list [symmetric] o_def)
apply ((rule has_type_cl_sub [THEN spec]) , (rule has_type_cl_sub [THEN spec]))
apply (simp add: eq_sym_conv)
apply (simp add: subst_comp_scheme_list [symmetric] o_def has_type_cl_sub eq_sym_conv)
apply (rule has_type_cl_sub [THEN spec])
apply (frule new_tv_W)
apply assumption
apply (drule conjunct1)
apply (frule new_tv_subst_scheme_list)
apply (rule new_tv_le)
apply (rule W_var_ge)
apply assumption
apply assumption
apply (erule thin_rl)
apply (erule allE)+
apply (drule sym)
apply (drule sym)
apply (erule thin_rl)
apply (erule thin_rl)
apply (erule (1) notE impE)
apply (erule (1) notE impE)
by assumption
case (LET e1 e2) thus ?case
apply (simp (no_asm) split: split_option_bind)
apply (intro strip)

apply (rename_tac S1 t1 m1 S2)
apply (rule_tac ?t1.0 = "$ S2 t1" in has_type.LETI)
  apply (simp (no_asm) add: o_def)
  apply (simp only: subst_comp_scheme_list [symmetric])
  apply (rule has_type_cl_sub [THEN spec])
  apply (drule_tac x = "A" in spec)
  apply (drule_tac x = "S1" in spec)
  apply (drule_tac x = "t1" in spec)
  apply (drule_tac x = "m1" in spec)
  apply (drule_tac x = "n" in spec)
  apply (erule (1) notE impE)
  apply (simp add: eq_sym_conv)
  apply (simp (no_asm) add: o_def)
  apply (simp only: subst_comp_scheme_list [symmetric])
  apply (rule gen_subst_commutates [symmetric, THEN subst])

```

```

apply (rule_tac [2] app_subst_Cons [THEN subst])
apply (erule_tac [2] thin_rl)
apply (drule_tac [2] x = "gen ($S1 A) t1 # $ S1 A" in spec)
apply (drule_tac [2] x = "S2" in spec)
apply (drule_tac [2] x = "t" in spec)
apply (drule_tac [2] x = "m" in spec)
apply (drule_tac [2] x = "m1" in spec)
apply (frule_tac [2] new_tv_W)
  prefer 2 apply (assumption)
  prefer 2
  apply (metis new_tv_Cons new_tv_compatible_W new_tv_compatible_gen new_tv_subst_scheme_list)
apply (rule weaken_A_Int_B_eq_empty)
apply (rule allI)
apply (intro strip)
apply (rule weaken_not_elem_A_minus_B)
  by (metis free_tv_W free_tv_gen_cons free_tv_le_new_tv new_tv_W)
qed

```

— Completeness of  $W$  w.r.t.  $has\_type$

**lemma**  $W\_complete\_lemma$  [rule\_format]:

$$\begin{aligned}
& \forall S' A t' n. \ \$S' A \vdash e :: t' \longrightarrow new\_tv\ n\ A \longrightarrow \\
& \quad (\exists S\ t. (\exists m. W\ e\ A\ n = Some\ (S, t, m)) \wedge \\
& \quad (\exists R. \ \$S' A = \$R\ (\$S\ A) \wedge t' = \$R\ t))
\end{aligned}$$

```

proof (induct e)
  case (Var) thus ?case
    apply (intro strip)
    apply (simp (no_asm) cong add: conj_cong)
    apply (erule has_type_casesE)
    apply (simp add: is_bound_typ_instance)
    apply (erule exE)
    apply (hypsubst)
    apply (rename_tac "S")
    apply (rule_tac x = "\lambda x. (if x < n then S' x else S (x - n))" in exI)
    apply (rule conjI)
      apply (simp add: new_if_subst_type_scheme_list)
      apply (simp (no_asm_simp) add: new_if_subst_type_scheme bound_typ_inst_composed_subst
[symmetric] new_tv_nth_nat_A o_def nth_subst
  del: bound_typ_inst_composed_subst)
  done
  case (Abs e) thus ?case
    apply (intro strip)
    apply (erule has_type_casesE)
    apply (erule_tac x = "\lambda x. if x=n then t1 else (S' x) " in allE)
    apply (erule_tac x = " (FVar n) #A" in allE)
    apply (erule_tac x = "t2" in allE)
    apply (erule_tac x = "Suc n" in allE)
    apply (bestsimp dest!: mk_scheme_injective cong: conj_cong split: split_option_bind)
  done
  case (App e1 e2) thus ?case

```

```

apply (intro strip)
apply (erule has_type_casesE)
apply (erule_tac x = "S'" in allE)
apply (erule_tac x = "A" in allE)
apply (erule_tac x = "t2 -> t'" in allE)
apply (erule_tac x = "n" in allE)
apply safe
apply (erule_tac x = "R" in allE)
apply (erule_tac x = "$ S A" in allE)
apply (erule_tac x = "t2" in allE)
apply (erule_tac x = "m" in allE)
apply simp
apply safe
  apply (blast intro: sym [THEN W_var_geD] new_tv_W [THEN conjunct1] new_tv_le new_tv_subst_schem

  apply (subgoal_tac "$ (λx. if x=ma then t' else (if x: (free_tv t - free_tv Sa) then
R x else Ra x)) ($ Sa t) = $ (λx. if x=ma then t' else (if x: (free_tv t - free_tv Sa)
then R x else Ra x)) (ta -> (TVar ma))")
  apply (rule_tac [2] t = "$ (λx. if x = ma then t' else (if x: (free_tv t - free_tv
Sa) then R x else Ra x)) ($ Sa t) " and s = " ($ Ra ta) -> t'" in ssubst)
  prefer 2 apply (simp (no_asm_simp) add: subst_comp_te) prefer 2
  apply (rule_tac [2] eq_free_eq_subst_te)
  prefer 2 apply (intro strip) prefer 2
  apply (subgoal_tac [2] "na ≠ ma")
  prefer 3 apply (best dest: new_tv_W sym [THEN W_var_geD] new_tv_not_free_tv new_tv_le)
  apply (case_tac [2] "na:free_tv Sa")

  apply (frule_tac [3] not_free_impl_id)
  prefer 3 apply (simp)

  apply (drule_tac [2] A1 = "$ S A" in trans [OF _ subst_comp_scheme_list])
  apply (drule_tac [2] eq_subst_scheme_list_eq_free)
  prefer 2 apply (fast intro: free_tv_W free_tv_le_new_tv dest: new_tv_W)
  prefer 2 apply (simp (no_asm_simp)) prefer 2
  apply (case_tac [2] "na:dom Sa")

  prefer 3 apply (simp add: dom_def)

  apply (rule_tac [2] eq_free_eq_subst_te)
  prefer 2 apply (intro strip) prefer 2
  apply (subgoal_tac [2] "nb ≠ ma")
  apply (frule_tac [3] new_tv_W) prefer 3 apply assumption
  apply (erule_tac [3] conjE)
  apply (drule_tac [3] new_tv_subst_scheme_list)
  prefer 3 apply (fast intro: new_tv_le dest: sym [THEN W_var_geD])
  prefer 3 apply (fastforce dest: new_tv_W new_tv_not_free_tv simp add: cod_def free_tv_subst)
  prefer 2 apply (fastforce simp add: cod_def free_tv_subst)
  prefer 2 apply (simp (no_asm)) prefer 2

```

```

apply (rule_tac [2] eq_free_eq_subst_te)
prefer 2 apply (intro strip) prefer 2
apply (subgoal_tac [2] "na ≠ ma")
  apply (frule_tac [3] new_tv_W) prefer 3 apply assumption
  apply (erule_tac [3] conjE)
  apply (drule_tac [3] sym [THEN W_var_geD])
  prefer 3 apply (fast dest: new_tv_le new_tv_subst_scheme_list new_tv_W new_tv_not_free_tv)
apply (case_tac [2] "na: free_tv t - free_tv Sa")

prefer 3
apply simp
apply fast

prefer 2 apply simp prefer 2
apply (drule_tac [2] A1 = "$ S A" and r = "$ R ($ S A)" in trans [OF _ subst_comp_scheme_list])
apply (drule_tac [2] eq_subst_scheme_list_eq_free)
  prefer 2
  apply (fast intro: free_tv_W free_tv_le_new_tv dest: new_tv_W)

prefer 2 apply (simp add: free_tv_subst dom_def)
apply (simp (no_asm_simp) split: split_option_bind)
apply safe
  apply (drule mgu_Some)
  apply fastforce

apply (drule mgu_mg, assumption)
apply (erule exE)
apply (rule_tac x = "Rb" in exI)
apply (rule conjI)
  apply (drule_tac [2] x = "ma" in fun_cong)
  prefer 2 apply (simp add: eq_sym_conv)
apply (simp (no_asm) add: subst_comp_scheme_list)
apply (simp (no_asm) add: subst_comp_scheme_list [symmetric])
apply (rule_tac A1 = "($ Sa ($ S A))" in trans [OF _ subst_comp_scheme_list [symmetric]])
apply (simp add: o_def eq_sym_conv)
apply (drule_tac s = "Some _" in sym)
apply (rule eq_free_eq_subst_scheme_list)
apply safe
apply (subgoal_tac "ma ≠ na")
  apply (frule_tac [2] new_tv_W) prefer 2 apply assumption
  apply (erule_tac [2] conjE)
  apply (drule_tac [2] new_tv_subst_scheme_list)
  prefer 2 apply (fast intro: new_tv_le dest: sym [THEN W_var_geD])
  apply (frule_tac [2] n = "m" in new_tv_W) prefer 2 apply assumption
  apply (erule_tac [2] conjE)
  apply (drule_tac [2] free_tv_app_subst_scheme_list [THEN subsetD])
  apply (tactic <
    (fast_tac (put_claset (claset_of @{theory_context Fun}) @{context}
      addDs [sym RS @{thm W_var_geD}, @{thm new_tv_le}, @{thm codD}],

```

```

    @{thm new_tv_not_free_tv}}] 2)>)
  apply (case_tac "na: free_tv t - free_tv Sa")

  prefer 2 apply simp apply blast

  apply simp
  apply (drule free_tv_app_subst_scheme_list [THEN subsetD])
  apply (fastforce dest: codD trans [OF _ subst_comp_scheme_list]
    eq_subst_scheme_list_eq_free
    simp add: free_tv_subst dom_def)
done
case (LET e1 e2) thus ?case
apply safe
  apply (erule has_type_casesE)
  apply (erule_tac x = "S" in allE)
  apply (erule_tac x = "A" in allE)
  apply (erule_tac x = "t1" in allE)
  apply (erule_tac x = "n" in allE)
  apply (erule (1) notE impE)
  apply (erule (1) notE impE)
  apply safe
  apply (simp (no_asm_simp))
  apply (erule_tac x = "R" in allE)
  apply (erule_tac x = "gen ($ S A) t # $ S A" in allE)
  apply (erule_tac x = "t'" in allE)
  apply (erule_tac x = "m" in allE)
  apply simp
  apply (drule mp)
  apply (rule has_type_le_env)
  apply assumption
  apply (simp (no_asm))
  apply (rule gen_bound_typ_instance)
  apply (drule mp)
  apply (frule new_tv_compatible_W)
  apply (rule sym)
  apply assumption
  apply (fast dest: new_tv_compatible_gen new_tv_subst_scheme_list new_tv_W)
  apply safe
  apply simp
  apply (rule_tac x = "Ra" in exI)
  apply (simp (no_asm) add: o_def subst_comp_scheme_list [symmetric])
done
qed

theorem W_complete:
  "[[] ⊢ e :: t' ⇒ ∃S t m. W e [] n = Some(S,t,m) ∧ (∃R. t' = $ R t)]"
by (metis W_complete_lemma app_subst_Nil new_tv_Nil)

end

```

## References

- [1] W. Naraschewski and T. Nipkow. Type inference verified: Algorithm W in Isabelle/HOL. In E. Giménez and C. Paulin-Mohring, editors, *Types for Proofs and Programs: Intl. Workshop TYPES '96*, volume 1512, pages 317–332, 1998.
- [2] W. Naraschewski and T. Nipkow. Type inference verified: Algorithm W in Isabelle/HOL. *Journal of Automated Reasoning*, 23:299–318, 1999.