

# MiniML

Wolfgang Naraschewski and Tobias Nipkow

December 12, 2009

## 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 (%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 --> P None) & (!s. res = Some s --> P(f s)))"
  unfolding option_bind_def
  by (rule option.split)
```

```
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

— new class for structures containing type variables
axclass type_struct < type

— 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
consts
  mk_scheme :: "typ => type_scheme"
primrec
  "mk_scheme (TVar n) = (FVar n)"
  "mk_scheme (t1 -> t2) = ((mk_scheme t1) ==> (mk_scheme t2))"

instance "typ"::type_struct ..
instance type_scheme::type_struct ..
instance list::(type_struct)type_struct ..
instance "fun"::(type,type_struct)type_struct ..

— free_tv s: the type variables occuring freely in the type structure s
consts
  free_tv :: "[’a::type_struct] => nat set"

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

primrec (free_tv_type_scheme)
  "free_tv (FVar m) = {m}"
  "free_tv (BVar m) = {}"
  "free_tv (S1 ==> S2) = (free_tv S1) Un (free_tv S2)"

```

```

primrec (free_tv_list)
  "free_tv [] = {}"
  "free_tv (x#l) = (free_tv x) Un (free_tv l)"

```

— executable version of `free_tv`: Implementation with lists

```

consts
  free_tv_ML :: "'a::type_struct] => nat list"

```

```

primrec (free_tv_ML_type_scheme)
  "free_tv_ML (FVar m) = [m]"
  "free_tv_ML (BVar m) = []"
  "free_tv_ML (S1 ==> S2) = (free_tv_ML S1) @ (free_tv_ML S2)"

```

```

primrec (free_tv_ML_list)
  "free_tv_ML [] = []"
  "free_tv_ML (x#l) = (free_tv_ML x) @ (free_tv_ML l)"

```

`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)"

```

— `bound_tv s`: the type variables occurring bound in the type structure `s`

```

consts
  bound_tv :: "'a::type_struct] => nat set"

```

```

primrec (bound_tv_type_scheme)
  "bound_tv (FVar m) = {}"
  "bound_tv (BVar m) = {m}"
  "bound_tv (S1 ==> S2) = (bound_tv S1) Un (bound_tv S2)"

```

```

primrec (bound_tv_list)
  "bound_tv [] = {}"
  "bound_tv (x#l) = (bound_tv x) Un (bound_tv l)"

```

— minimal new free / bound variable

```

consts
  min_new_bound_tv :: "'a::type_struct => nat"

```

```

primrec (min_new_bound_tv_type_scheme)
  "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)"

```

— substitutions

— type variable substitution

**types** *subst* = "nat => typ"

— identity

**definition**

```
id_subst :: subst where
  "id_subst = (%n. TVar n)"
```

— extension of substitution to type structures

**consts**

```
app_subst :: "[subst, 'a::type_struct] => 'a::type_struct" ("$$")
```

**primrec** (*app\_subst\_typ*)

```
app_subst_TVar: "$ S (TVar n) = S n"
app_subst_Fun:  "$ S (t1 -> t2) = ($ S t1) -> ($ S t2)"
```

**primrec** (*app\_subst\_type\_scheme*)

```
"$ S (FVar n) = mk_scheme (S n)"
"$ S (BVar n) = (BVar n)"
"$ S (sch1 ==> sch2) = ($ S sch1) ==> ($ S sch2)"
```

**defs** (*overloaded*)

```
app_subst_list: "$ S == map ($ S)"
```

— 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)))"
```

**defs** (*overloaded*)

```
free_tv_subst: "free_tv S == (dom S) Un (cod S)"
```

— unification algorithm mgu

**consts**

```
mgu :: "[typ,typ] => subst option"
```

**axioms**

```
mgu_eq: "mgu t1 t2 = Some U ==> $U t1 = $U t2"
mgu_mg: "[| (mgu t1 t2) = Some U; $$ t1 = $$ t2 |] ==> ? R. S = $R o U"
mgu_Some: "$S t1 = $$ t2 ==> ? U. mgu t1 t2 = Some U"
mgu_free: "mgu t1 t2 = Some U ==> (free_tv U) <= (free_tv t1) Un (free_tv t2)"
```

```

declare mgu_eq [simp] mgu_mg [simp] mgu_free [simp]

lemma mk_scheme_Fun [rule_format]: "mk_scheme t = sch1 ==> sch2 ==> (? t1 t2. sch1 =
mk_scheme t1 & sch2 = mk_scheme t2)"
apply (induct_tac "t")
apply (simp (no_asm))
apply simp
apply fast
done

lemma mk_scheme_injective [rule_format]: "!t'. mk_scheme t = mk_scheme t' ==> t=t'"
apply (induct_tac "t")
  apply (rule allI)
  apply (induct_tac "t'")
    apply (simp (no_asm))
    apply simp
  apply (rule allI)
  apply (induct_tac "t'")
    apply (simp (no_asm))
    apply simp
done

lemma free_tv_mk_scheme: "free_tv (mk_scheme t) = free_tv t"
apply (induct_tac "t")
apply (simp_all (no_asm_simp))
done

declare free_tv_mk_scheme [simp]

lemma subst_mk_scheme: "$ S (mk_scheme t) = mk_scheme ($ S t)"
apply (induct_tac "t")
apply (simp_all (no_asm_simp))
done

declare subst_mk_scheme [simp]

— constructor laws for app_subst

lemma app_subst_Nil:
  "$ S [] = []"

apply (unfold app_subst_list)
apply (simp (no_asm))
done

lemma app_subst_Cons:

```

```

    "$ S (x#1) = ($ S x)#($ S 1)"
  apply (unfold app_subst_list)
  apply (simp (no_asm))
done

declare app_subst_Nil [simp] app_subst_Cons [simp]

— constructor laws for new_tv

lemma new_tv_TVar:
  "new_tv n (TVar m) = (m<n)"

  apply (unfold new_tv_def)
  apply (fastsimp)
done

lemma new_tv_FVar:
  "new_tv n (FVar m) = (m<n)"
  apply (unfold new_tv_def)
  apply (fastsimp)
done

lemma new_tv_BVar:
  "new_tv n (BVar m) = True"
  apply (unfold new_tv_def)
  apply (simp (no_asm))
done

lemma new_tv_Fun:
  "new_tv n (t1 -> t2) = (new_tv n t1 & new_tv n t2)"
  apply (unfold new_tv_def)
  apply (fastsimp)
done

lemma new_tv_Fun2:
  "new_tv n (t1 ==> t2) = (new_tv n t1 & new_tv n t2)"
  apply (unfold new_tv_def)
  apply (fastsimp)
done

lemma new_tv_Nil:
  "new_tv n []"
  apply (unfold new_tv_def)
  apply (simp (no_asm))
done

lemma new_tv_Cons:
  "new_tv n (x#l) = (new_tv n x & new_tv n l)"

```

```

apply (unfold new_tv_def)
apply (fastsimp)
done

lemma new_tv_TVar_subst: "new_tv n TVar"
apply (unfold new_tv_def)
apply (intro strip)
apply (simp add: free_tv_subst dom_def cod_def)
done

declare
  new_tv_TVar [simp] new_tv_FVar [simp] new_tv_BVar [simp]
  new_tv_Fun [simp] new_tv_Fun2 [simp] new_tv_Nil [simp]
  new_tv_Cons [simp] new_tv_TVar_subst [simp]

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 [simp]: "new_tv n (sch::type_scheme) ==>
  $(%k. if k<n then S k else S' k) sch = $S sch"
  by (induct sch) simp_all

lemma new_if_subst_type_scheme_list [simp]: "new_tv n (A::type_scheme list) ==>
  $(%k. if k<n then S k else S' k) A = $S A"
  by (induct A) simp_all

— 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 [rule_format, simp]:
  "!n. n < length A --> x : free_tv (A!n) --> x : free_tv A"
apply (induct A)
apply simp
apply (rule allI)
apply (induct_tac n)
apply simp
apply simp
done

```

```

lemma free_tv_nth_nat_A [rule_format]:
  "!nat. nat < length A --> x : free_tv (A!nat) --> x : free_tv A"
apply (induct A)
apply simp
apply (rule allI)
apply (induct_tac nat)
apply (intro strip)
apply simp
apply simp
done

```

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:
  "(!x:free_tv t. (S x) = (S' x)) ==> $ S (t::typ) = $ S' t"
  by (induct t) simp_all

```

```

lemma eq_free_eq_subst_te: "(!n. n:(free_tv t) --> S1 n = S2 n) ==> $ S1 (t::typ) = $ S2 t"
apply (rule typ_substitutions_only_on_free_variables)
apply simp
done

```

```

lemma scheme_substitutions_only_on_free_variables:
  "(!x:free_tv sch. (S x) = (S' x)) ==> $ S (sch::type_scheme) = $ S' sch"
  by (induct sch) simp_all

```

```

lemma eq_free_eq_subst_type_scheme:
  "(!n. n:(free_tv sch) --> S1 n = S2 n) ==> $ S1 (sch::type_scheme) = $ S2 sch"
apply (rule scheme_substitutions_only_on_free_variables)
apply simp
done

```

```

lemma eq_free_eq_subst_scheme_list:
  "(!n. n:(free_tv A) --> S1 n = S2 n) ==> $S1 (A::type_scheme list) = $S2 A"
proof (induct A)
  case Nil then show ?case by fastsimp
next
  case Cons then show ?case by (fastsimp intro: eq_free_eq_subst_type_scheme)
qed

```

```

lemma weaken_asm_Un: "(!x:A. (P x)) --> Q ==> (!x:A Un B. (P x)) --> Q"
  by fast

```

```

lemma scheme_list_substitutions_only_on_free_variables [rule_format]:
  "(!x:free_tv A. (S x) = (S' x)) --> $ S (A::type_scheme list) = $ S' A"
apply (induct_tac A)
apply simp
apply simp

```

```

apply (rule weaken_asm_Un)
apply (intro strip)
apply (erule scheme_substitutions_only_on_free_variables)
done

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

lemma eq_subst_type_scheme_eq_free [rule_format]:
  "$ S1 (sch::type_scheme) = $ S2 sch --> n:(free_tv sch) --> S1 n = S2 n"
  apply (induct_tac "sch")

apply simp

apply (intro strip)
apply (erule mk_scheme_injective)
apply simp

apply simp
done

lemma eq_subst_scheme_list_eq_free:
  "$S1 (A::type_scheme list) = $S2 A ==> n:(free_tv A) ==> S1 n = S2 n"
proof (induct A)
  case Nil
  then show ?case by fastsimp
next
  case Cons
  then show ?case by (fastsimp intro: eq_subst_type_scheme_eq_free)
qed

lemma codD: "v : cod S ==> v : free_tv S"
  unfolding free_tv_subst by blast

lemma not_free_impl_id: "x ~: free_tv S ==> 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 [simp]:
  "[| v : free_tv(S n); v~=n |] ==> v : cod S"
  apply (unfold dom_def cod_def UNION_def Bex_def)
  apply (simp (no_asm))
  apply (safe intro!: exI conjI notI)
  prefer 2 apply (assumption)
  apply simp
done

```

```

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

lemma free_tv_app_subst_te: "free_tv ($ S (t::typ)) <= cod S Un 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 fastsimp
qed

lemma free_tv_app_subst_type_scheme:
  "free_tv ($ S (sch::type_scheme)) <= cod S Un 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 fastsimp
qed

lemma free_tv_app_subst_scheme_list: "free_tv ($ S (A::type_scheme list)) <= cod S Un
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 fastsimp
qed

lemma free_tv_comp_subst:
  "free_tv (%u::nat. $ s1 (s2 u) :: typ) <=
  free_tv s1 Un 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) <= free_tv S1 Un 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) <= 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) <= 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) <= 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 = ((!m. n <= m --> (S m = TVar m)) &
    (! l. l < n --> new_tv n (S l) ))"

apply (unfold new_tv_def)
apply (safe)

  apply (fastsimp dest: leD simp add: free_tv_subst dom_def)
  apply (subgoal_tac "m:cod S | S l = TVar l")

```

```

apply safe
  apply (fastsimp dest: UnI2 simp add: free_tv_subst)
  apply (drule_tac P = "%x. m:free_tv x" in subst , assumption)
  apply simp
  apply (fastsimp simp add: free_tv_subst cod_def dom_def)

apply (unfold free_tv_subst cod_def dom_def)
apply safe
apply (metis not_leE)+
done

lemma new_tv_list: "new_tv n x = (!y:set x. new_tv n y)"
  by (induct x) simp_all

— substitution affects only variables occurring freely
lemma subst_te_new_tv [simp]:
  "new_tv n (t::typ) --> $(%x. if x=n then t' else S x) t = $S t"
  by (induct t) simp_all

lemma subst_te_new_type_scheme [simp]:
  "new_tv n (sch::type_scheme) ==> $(%x. 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) ==> $(%x. if x=n then t else S x) A = $S A"
  by (induct A) simp_all

— all greater variables are also new
lemma new_tv_le:
  "n<=m ==> new_tv n t ==> new_tv m t"
  apply (unfold new_tv_def)
  apply safe
  apply (drule spec)
  apply (erule (1) notE impE)
  apply (simp (no_asm))
  done

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

lemma new_tv_typ_le: "n<=m ==> new_tv n (t::typ) ==> new_tv m t"
  by (simp add: new_tv_le)

lemma new_scheme_list_le: "n<=m ==> new_tv n (A::type_scheme list) ==> new_tv m A"
  by (simp add: new_tv_le)

lemma new_tv_subst_le: "n<=m ==> new_tv n (S::subst) ==> new_tv m S"
  by (simp add: 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 (induct t) (auto simp add: new_tv_subst)

lemma new_tv_subst_type_scheme [rule_format, simp]:
  "new_tv n S --> new_tv n (sch :: type_scheme) --> new_tv n ($ S sch)"
  apply (induct sch)
  apply (simp_all)
  apply (unfold new_tv_def)
  apply (simp (no_asm) add: free_tv_subst dom_def cod_def)
  apply (intro strip)
  apply (case_tac "S nat = TVar nat")
  apply simp
  apply (drule_tac x = "m" in spec)
  apply fast
  done

lemma new_tv_subst_scheme_list [simp]:
  "new_tv n S ==> new_tv n (A :: type_scheme list) ==> new_tv n ($ S A)"
  by (induct A) auto

lemma new_tv_Suc_list: "new_tv n A --> new_tv (Suc n) ((TVar n)#A)"
  by (simp add: new_tv_list)

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' ==> new_tv n A ==> new_tv n A'"
  unfolding new_tv_def by simp

lemma new_tv_nth_nat_A [rule_format]:
  "!nat. nat < length A --> new_tv n A --> (new_tv n (A!nat))"
  apply (unfold new_tv_def)
  apply (induct A)
  apply simp
  apply (rule allI)
  apply (induct_tac "nat")
  apply (intro strip)
  apply simp

```

```

apply (simp (no_asm))
done

```

— composition of substitutions preserves `new_tv` proposition

```

lemma new_tv_subst_comp_1 [simp]:
  "[| new_tv n (S::subst); new_tv n R |] ==> new_tv n (($ R) o S)"
  by (simp add: new_tv_subst)

```

```

lemma new_tv_subst_comp_2 [simp]:
  "[| new_tv n (S::subst); new_tv n R |] ==> new_tv n (%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 [simp]:
  "new_tv n A ==> n~:(free_tv A)"
  by (auto simp add: new_tv_def)

```

```

lemma fresh_variable_types [simp]: "!!t::typ. ? n. (new_tv n t)"
apply (unfold new_tv_def)
apply (induct_tac t)
apply (rule_tac x = "Suc nat" in exI)
apply (simp (no_asm_simp))
apply (erule exE)+
apply (rename_tac "n'")
apply (rule_tac x = "max n n'" in exI)
apply (simp add: less_max_iff_disj)
done

```

```

lemma fresh_variable_type_schemes [simp]:
  "!!sch::type_scheme. ? n. (new_tv n sch)"
apply (unfold new_tv_def)
apply (induct_tac sch)
apply (rule_tac x = "Suc nat" in exI)
apply (simp (no_asm))
apply (rule_tac x = "Suc nat" in exI)
apply (simp (no_asm))
apply (erule exE)+
apply (rename_tac "n'")
apply (rule_tac x = "max n n'" in exI)
apply (simp add: less_max_iff_disj)
done

```

```

lemma fresh_variable_type_scheme_lists [simp]:
  "!!A::type_scheme list. ? n. (new_tv n A)"
apply (induct_tac A)
apply (simp (no_asm))
apply (simp (no_asm))
apply (erule exE)

```

```

apply (cut_tac sch = "a" in fresh_variable_type_schemes)
apply (erule exE)
apply (rename_tac "n'")
apply (rule_tac x = " (max n n') " in exI)
apply (subgoal_tac "n <= (max n n') ")
apply (subgoal_tac "n' <= (max n n') ")
apply (fast dest: new_tv_le)
apply (simp_all add: le_max_iff_disj)
done

```

```

lemma make_one_new_out_of_two:
  "[| ? n1. (new_tv n1 x); ? n2. (new_tv n2 y)|] ==> ? n. (new_tv n x) & (new_tv n y)"
apply (erule exE)+
apply (rename_tac "n1" "n2")
apply (rule_tac x = " (max n1 n2) " in exI)
apply (subgoal_tac "n1 <= max n1 n2")
apply (subgoal_tac "n2 <= max n1 n2")
apply (fast dest: new_tv_le)
apply (simp_all (no_asm) add: le_max_iff_disj)
done

```

```

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')"
apply (cut_tac t = "t" in fresh_variable_types)
apply (cut_tac t = "t'" in fresh_variable_types)
apply (drule make_one_new_out_of_two)
apply assumption
apply (erule_tac V = "? n. new_tv n t'" in thin_rl)
apply (cut_tac A = "A" in fresh_variable_type_scheme_lists)
apply (cut_tac A = "A'" in fresh_variable_type_scheme_lists)
apply (rotate_tac 1)
apply (drule make_one_new_out_of_two)
apply assumption
apply (erule_tac V = "? n. new_tv n A'" in thin_rl)
apply (erule exE)+
apply (rename_tac n2 n1)
apply (rule_tac x = " (max n1 n2) " in exI)
apply (unfold new_tv_def)
apply (simp (no_asm) add: less_max_iff_disj)
apply blast
done

```

— 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"
apply (unfold new_tv_def)
apply (fast dest: mgu_free)
done

```

```
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 = (%t::typ. t)"
  apply (unfold id_subst_def)
  apply (rule ext)
  apply (induct_tac "t")
  apply simp_all
  done
```

```
lemma app_subst_id_type_scheme [simp]:
  "$ id_subst = (%sch::type_scheme. sch)"
  apply (unfold id_subst_def)
  apply (rule ext)
  apply (induct_tac "sch")
  apply simp_all
  done
```

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

```
lemma app_subst_id_tel [simp]:
  "$ id_subst = (%A::type_scheme list. A)"
  apply (unfold app_subst_list)
  apply (rule ext)
  apply (induct_tac "A")
  apply simp_all
  done
```

```
lemma id_subst_sch [simp]:
  fixes sch :: type_scheme
  shows "$ id_subst sch = sch"
  by (induct sch) (simp_all add: id_subst_def)
```

```
lemma id_subst_A [simp]:
```

```

fixes A :: "type_scheme list"
shows "$ id_subst A = A"
by (induct A) simp_all

— 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) = $ (%x. $ R (S x) ) t"
  by (induct t) simp_all

lemma subst_comp_type_scheme:
  "$ R ($ S sch::type_scheme) = $ (%x. $ R (S x) ) sch"
  by (induct sch) simp_all

lemma subst_comp_scheme_list:
  "$ R ($ S A::type_scheme list) = $ (%x. $ R (S x)) A"
  unfolding app_subst_list
  proof (induct A)
    case Nil then show ?case by simp
  next
    case (Cons x xl)
    then show ?case by (simp add: subst_comp_type_scheme)
  qed

lemma subst_id_on_type_scheme_list':
  fixes A :: "type_scheme list"
  shows "!x : free_tv A. S x = (TVar x) ==> $ S A = $ id_subst A"
  apply (rule scheme_list_substitutions_only_on_free_variables)
  apply (simp add: id_subst_def)
  done

lemma subst_id_on_type_scheme_list:
  fixes A :: "type_scheme list"
  shows "!x : free_tv A. S x = (TVar x) ==> $ S A = A"
  apply (subst subst_id_on_type_scheme_list')
  apply assumption
  apply simp
  done

lemma nth_subst [rule_format]:
  "!n. n < length A --> ($ S A)!n = $S (A!n)"
  apply (induct A)
  apply simp
  apply (rule allI)
  apply (rename_tac "n1")
  apply (induct_tac "n1")
  apply simp
  apply simp

```

done

end

### 3 Instances of type schemes

theory *Instance*

imports *Type*

begin

consts

*bound\_typ\_inst* :: "[subst, type\_scheme] => typ"

primrec

"*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*))"

consts

*bound\_scheme\_inst* :: "[nat => type\_scheme, type\_scheme] => type\_scheme"

primrec

"*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 = (? S. t = (*bound\_typ\_inst* *S* sch))"

instantiation *type\_scheme* :: ord

begin

definition

*le\_type\_scheme\_def*: "sch' <= (sch :: type\_scheme)  $\longleftrightarrow$  (!t. t <| sch' --> t <| sch)"

definition

"(sch' < (sch :: type\_scheme))  $\longleftrightarrow$  sch'  $\leq$  sch  $\wedge$  sch'  $\neq$  sch"

instance ..

end

consts

*subst\_to\_scheme* :: "[nat => type\_scheme, typ] => type\_scheme"

primrec

"*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 ($S o R) ($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 o 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 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 = "%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 (%k. if n <= k then BVar (k - n) else FVar k)
  (bound_typ_inst (%k. 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 (%k. if n <= k then BVar (k - n) else FVar k) t =
  subst_to_scheme (%k. if n <= k then BVar (k - n) else FVar k) t'"
by blast

```

```

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

```

```

lemma le_type_scheme_def2:
  fixes sch sch' :: type_scheme
  shows "(sch' <= sch) = (? B. sch' = bound_scheme_inst B 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 = "? n. new_tv n sch'" in thin_rl)
apply (erule exE)
apply (erule allE)
apply (drule mp)
apply (rule_tac x = " (%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 (%k. if n <= k then BVar (k - n) else FVar k))
o S" in exI)
apply (simp (no_asm_simp) add: aux2)
apply safe
apply (rule_tac x = "%n. 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) <= 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 <= sch' # B) = (sch <= (sch'::type_scheme) & A <= 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 ==> $S t <| $S sch"
apply (unfold is_bound_typ_instance)
apply (erule exE)
apply (rename_tac "SA")
apply simp
apply (rule_tac x = "$S o SA" in exI)
apply simp
done

```

```

lemma S_compatible_le_scheme:
  fixes sch sch' :: type_scheme
  shows "sch' <= sch ==> $S sch' <= $ 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' <= A ==> $S A' <= $ 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 <= sch' |] ==> t <| sch'"
  unfolding le_type_scheme_def by blast

```

```

lemma le_type_scheme_refl [iff]: "sch <= (sch::type_scheme)"
  unfolding le_type_scheme_def by blast

```

```

lemma le_env_refl [iff]: "A <= (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 = "%a. 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 fastsimp
done

lemma Fun_le_FunD:
  "(sch1 ==> sch2 <= sch1' ==> sch2') ==> sch1 <= sch1' & sch2 <= sch2'"
  unfolding le_type_scheme_def is_bound_typ_instance by fastsimp

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

types ctxt = "type_scheme list"

consts
  gen :: "[ctxt, typ] => type_scheme"
primrec
  "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

consts
  gen_ML_aux :: "[nat list, typ] => type_scheme"
primrec
  "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)"

consts
  gen_ML :: "[ctxt, typ] => type_scheme"
defs
  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)  $\implies$  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 fastsimp

lemma bound_tv_gen [simp]:
  "bound_tv (gen A t1) = (free_tv t1) - (free_tv A)"
  apply (induct t1)
  apply (simp (no_asm))
  apply (case_tac "nat : free_tv A")
  apply (simp (no_asm_simp))
  apply (simp (no_asm_simp))
  apply fast
  apply (simp (no_asm_simp))
  apply fast
  done

lemma new_tv_compatible_gen: "new_tv n t  $\implies$  new_tv n (gen A t)"
  by (induct 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_commutes [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 (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 = " (%a. bound_typ_inst R (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 <= free_tv A ==> gen A t <= gen B t"
  apply (unfold le_type_scheme_def is_bound_typ_instance)
  apply safe
  apply (rename_tac "S")
  apply (rule_tac x = "%b. if b:free_tv A then TVar b else S b" in exI)
  apply (induct_tac "t")
  apply fastsimp
  apply simp
done

lemma gen_t_le_gen_alpha_t [rule_format, simp]:
  "new_tv n A -->
  gen A t <= gen A ($ (%x. TVar (if x : free_tv A then x 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 = " (%x. S (if n <= x then x - n else x))" in exI)
  apply (induct t)
  apply (simp (no_asm))
  apply (case_tac "nat : free_tv A")
  apply (simp (no_asm_simp))
  apply (simp (no_asm_simp))
  apply (subgoal_tac "n <= 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 :: "[ctxt, 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:
  "!!t::typ. $(%n. 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

declare s'_t_equals_s_t [simp]

lemma s'_a_equals_s_a:
  "!!A::type_scheme list. $(%n. 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

declare s'_a_equals_s_a [simp]

```

```

lemma replace_s_by_s':
  "$(%n. if n : (free_tv A) Un (free_tv t) then S n else TVar n) A |-
    e :: $(%n. 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 = "%A. A |- e :: $S t" in ssubst)
apply (rule s'_a_equals_s_a [symmetric])
apply (rule_tac P = "%t. $ (%n. if n : free_tv A Un free_tv (?t2 S) then S n else TVar
n) A |- e :: t" in ssubst)
apply (rule s'_t_equals_s_t [symmetric])
apply simp
done

lemma alpha_A':
  "!!A::type_scheme list. $ (%x. 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. $ (%x. 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 ($ (%x. TVar (alpha x)) t)"
apply (induct_tac "t")
apply (simp_all)
done

lemma S_o_alpha_typ':
  "$ (%u. (S o alpha) u) (t::typ) = $ S ($ (%x. 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 ($ (%x. 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 ($ (%x. 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.
  $ (%n. if n : free_tv A Un free_tv t then S n else TVar n) A =
  $ ((%x. if x : free_tv A Un free_tv t then S x else TVar x) o
    (%x. 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 (%n. 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':
  "!!(A::type_scheme list) (t::typ).
  cod (%n. if n : free_tv A Un free_tv t then S n else TVar n) <=
  free_tv ($ S A) 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':
  "!!(A::type_scheme list) (t::typ).
  free_tv (%n. if n : free_tv A Un free_tv t then S n else TVar n) <=
  free_tv A Un free_tv ($ S A) Un free_tv t 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: "!!t1::typ.
  (free_tv ($ (%x. TVar (if x : free_tv A then x else n + x)) t1) - free_tv A) <=
  {x. ? 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: "!!t::typ. new_tv n t ==> {x. ? y. x = n + y} Int
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: "!!sch::type_scheme. new_tv n sch ==> {x. ? y.
x = n + y} Int 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: "!A::type_scheme list. new_tv n A --> {x.
? y. x = n + y} Int 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)]:
  "new_tv n A --> gen A t <= gen A ($ (%x. TVar (if x : free_tv A then x 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 = " (%x. S (if n <= x then x - n else x))" in exI)
apply (induct_tac t)
apply (simp (no_asm))
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 |- e::t ==> !B. A <= B --> B |- e::t"
apply (erule has_type.induct)
  apply (simp (no_asm) add: le_env_def)
  apply (fastsimp 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 = "$ ((%x. if x : free_tv A Un free_tv t then S x else TVar x) o
(%x. if x : free_tv A then x else n + x)) t1" in has_type.LETI)
  apply (drule_tac x = " (%x. if x : free_tv A Un free_tv t then S x else TVar x) o (%x.
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_commutates)
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 del: full_SetCompr_eq)
  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

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

— type inference algorithm W
consts W :: "[expr, ctxt, nat] => result_W"

primrec
  "W (Var i) A n =
    (if i < length A then Some( id_subst,

```

```

        bound_typ_inst (%b. 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]
declare less_imp_le [simp del] — the combination loops

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 [rule_format (no_asm)]:
"!A n S t m. W e A n = Some (S,t,m) --> n<=m"
apply (induct_tac "e")

apply (simp (no_asm) split add: split_option_bind)

apply (simp (no_asm) split add: split_option_bind)
apply (fast dest: Suc_leD)

apply (simp (no_asm) split add: split_option_bind)
apply (blast intro: le_SucI le_trans)

apply (simp (no_asm) split add: split_option_bind)
apply (blast intro: le_trans)
done

declare W_var_ge [simp]

lemma W_var_geD:
"Some (S,t,m) = W e A n ==> n<=m"
apply (simp add: eq_sym_conv)

```

done

lemma new\_tv\_compatible\_W:

```
"new_tv n A ==> Some (S,t,m) = W e A n ==> new_tv m A"
apply (drule W_var_geD)
apply (rule new_scheme_list_le)
apply assumption
apply assumption
done
```

lemma new\_tv\_bound\_typ\_inst\_sch [rule\_format (no\_asm)]:

```
"new_tv n sch --> new_tv (n + (min_new_bound_tv sch)) (bound_typ_inst (%b. TVar (b +
n)) sch)"
apply (induct_tac "sch")
  apply simp
  apply (simp add: add_commute)
apply (intro strip)
apply simp
apply (erule conjE)
apply (rule conjI)
  apply (rule new_tv_le)
  prefer 2 apply (assumption)
  apply (rule add_le_mono)
  apply (simp (no_asm))
  apply (simp (no_asm) add: max_def)
apply (rule new_tv_le)
  prefer 2 apply (assumption)
  apply (rule add_le_mono)
  apply (simp (no_asm))
  apply (simp (no_asm) add: max_def)
done
```

declare new\_tv\_bound\_typ\_inst\_sch [simp]

— resulting type variable is new

lemma new\_tv\_W [rule\_format (no\_asm)]:

```
"!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"
apply (induct_tac "e")
```

```
apply (simp (no_asm) split add: split_option_bind)
apply (intro strip)
apply (drule new_tv_nth_nat_A)
apply assumption
apply (simp (no_asm_simp))
```

```
apply (simp (no_asm) add: new_tv_subst new_tv_Suc_list split add: split_option_bind)
apply (intro strip)
apply (erule_tac x = "Suc n" in allE)
```

```

apply (erule_tac x = " (FVar n) #A" in allE)
apply (fastsimp simp add: new_tv_subst new_tv_Suc_list)

apply (simp (no_asm) split add: split_option_bind)
apply (intro strip)
apply (rename_tac S1 t1 n1 S2 t2 n2 S3)
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 (erule_tac x = "n1" in allE)
apply (erule_tac x = "n1" in allE)
apply (simp add: eq_sym_conv del: all_simps)
apply (erule_tac x = "$S1 A" in allE)
apply (erule_tac x = "S2" in allE)
apply (erule_tac x = "t2" in allE)
apply (erule_tac x = "n2" in allE)
apply (simp add: o_def rotate_Some)
apply (rule conjI)
apply (rule new_tv_subst_comp_2)
apply (rule new_tv_subst_comp_2)
apply (rule lessI [THEN less_imp_le, THEN new_tv_le])
apply (erule_tac n = "n1" in new_tv_subst_le)
apply (simp add: rotate_Some)
apply (simp (no_asm_simp))
apply (fast dest: W_var_geD intro: new_scheme_list_le new_tv_subst_scheme_list lessI [THEN
less_imp_le [THEN new_tv_subst_le]])
apply (erule sym [THEN mgu_new])
apply (blast dest!: W_var_geD
      intro: lessI [THEN less_imp_le, THEN new_tv_le] new_tv_subst_te
            new_tv_subst_scheme_list new_scheme_list_le new_tv_le)

apply (erule impE)
apply (blast dest: W_var_geD intro: new_tv_subst_scheme_list new_scheme_list_le new_tv_le)
apply clarsimp

apply (rule lessI [THEN new_tv_subst_var])
apply (erule sym [THEN mgu_new])
apply (blast dest!: W_var_geD
      intro: lessI [THEN less_imp_le, THEN new_tv_le] new_tv_subst_te
            new_tv_subst_scheme_list new_scheme_list_le new_tv_le)

apply (erule impE)
apply (blast dest: W_var_geD intro: new_tv_subst_scheme_list new_scheme_list_le new_tv_le)
apply clarsimp

— 41: case LET e1 e2
apply (simp (no_asm) split add: split_option_bind)
apply (intro strip)

```

```

apply (erule allE,erule allE,erule allE,erule allE,erule allE, erule impE, assumption,
erule impE, assumption)
apply (erule conjE)
apply (erule allE,erule allE,erule allE,erule allE,erule allE, erule impE, erule_tac [2]
notE impE, tactic "assume_tac 2")
apply (simp only: new_tv_def)
apply (simp (no_asm_simp))
apply (drule W_var_ge)+
apply (rule allI)
apply (intro strip)
apply (simp only: free_tv_subst)
apply (drule free_tv_app_subst_scheme_list [THEN subsetD])
apply (best elim: less_le_trans)
apply (erule conjE)
apply (rule conjI)
apply (simp only: o_def)
apply (rule new_tv_subst_comp_2)
apply (erule W_var_ge [THEN new_tv_subst_le])
apply assumption
apply assumption
apply assumption
done

```

```

lemma free_tv_bound_typ_inst1 [rule_format (no_asm)]:
  "(v ~: free_tv sch) --> (v : free_tv (bound_typ_inst (TVar o S) sch)) --> (? x. v =
S x)"
apply (induct_tac "sch")
apply simp
apply simp
apply (intro strip)
apply (rule exI)
apply (rule refl)
apply simp
done

```

```

declare free_tv_bound_typ_inst1 [simp]

```

```

lemma free_tv_W [rule_format (no_asm)]:
  "!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"
apply (induct e)

```

```

apply (simp (no_asm) add: free_tv_subst split add: split_option_bind)
apply (intro strip)
apply (case_tac "v : free_tv (A!nat) ")
  apply simp
apply (drule free_tv_bound_typ_inst1)
  apply (simp (no_asm) add: o_def)

```

```

apply (erule exE)
apply simp

apply (simp add: free_tv_subst split add: 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)

apply (simp (no_asm) split add: 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 (fastsimp dest: free_tv_comp_subst [THEN subsetD] sym [THEN mgu_free] codD free_tv_app_subst
[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 (fastsimp 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)

apply (simp (no_asm) split add: 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

lemma weaken_A_Int_B_eq_empty: "(!x. x : A --> x ~: B) ==> A Int B = {}"
apply fast
done

lemma weaken_not_elem_A_minus_B: "x ~: A | x : B ==> x ~: A - B"
apply fast
done

— correctness of W with respect to has_type
lemma W_correct_lemma [rule_format (no_asm)]: "!A S t m n . new_tv n A --> Some (S,t,m)
= W e A n --> $S A |- e :: t"
apply (induct_tac "e")

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)

apply (simp add: app_subst_list split add: 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_scheme_list_le])
apply (drule sym)
apply (erule allE)+
apply (erule impE)
apply (erule_tac [2] notE impE, tactic "assume_tac 2")
apply (simp (no_asm_simp))
apply assumption

apply (simp (no_asm) split add: 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
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_scheme_list_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 thin_rl)
apply (erule (1) notE impE)
apply (erule (1) notE impE)
apply assumption

apply (simp (no_asm) split add: 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_commutes [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)
  apply (drule_tac [2] conjunct1)
  apply (frule_tac [2] new_tv_subst_scheme_list)
    apply (rule_tac [2] new_scheme_list_le)
      apply (rule_tac [2] W_var_ge)
        prefer 2 apply (assumption)
      prefer 2 apply (assumption)
  apply (erule_tac [2] impE)
    apply (rule_tac [2] A = "$ S1 A" in new_tv_only_depends_on_free_tv_scheme_list)
      prefer 2 apply simp
      apply (fast)
    prefer 2 apply (assumption)
  prefer 2 apply simp
apply (rule weaken_A_Int_B_eq_empty)
apply (rule allI)
apply (intro strip)
apply (rule weaken_not_elem_A_minus_B)
apply (case_tac "x < m1")
  apply (rule disjI2)
  apply (rule free_tv_gen_cons [THEN subst])
  apply (rule free_tv_W)
    apply assumption
    apply simp
  apply assumption
apply (rule disjI1)
apply (drule new_tv_W)
apply assumption
apply (drule conjunct2)
apply (rule new_tv_not_free_tv)
apply (rule new_tv_le)
  prefer 2 apply (assumption)
apply (simp add: linorder_not_less)
done

```

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

**lemma**  $W\_complete\_lemma$  [rule\_format (no\_asm)]:

```
"ALL S' A t' n. $S' A |- e :: t' --> new_tv n A -->
  (EX S t. (EX m. W e A n = Some (S,t,m)) &
    (EX R. $S' A = $R ($S A) & t' = $R t))"
```

**apply** (induct e)

```
  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 = "%x. (if x < n then S' x else S (x - n))" in exI)
  apply (rule conjI)
  apply (simp (no_asm_simp))
  apply (simp (no_asm_simp) add: bound_typ_inst_composed_subst [symmetric] new_tv_nth_nat_A
o_def nth_subst
                                del: bound_typ_inst_composed_subst)
```

```
  apply (intro strip)
  apply (erule has_type_casesE)
  apply (erule_tac x = "%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)
```

```
  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_scheme_list_le
new_tv_subst_scheme_list)
```

```

apply (subgoal_tac "$ (%x. if x=ma then t' else (if x: (free_tv t - free_tv Sa) then R
x else Ra x)) ($ Sa t) = $ (%x. 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 = "$ (%x. 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_scheme_list_le dest: sym [THEN W_var_geD])
  prefer 3 apply (fastsimp dest: new_tv_W new_tv_not_free_tv simp add: cod_def free_tv_subst)
  prefer 2 apply (fastsimp 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_scheme_list_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 add: split_option_bind)
apply safe
apply (drule mgu_Some)
apply fastsimp

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 ?X" 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_scheme_list_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 (global_claset_of @{theory Fun} addDs [sym RS thm "W_var_geD", thm "new_scheme_list_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 (fastsimp dest: codD trans [OF _ subst_comp_scheme_list]
    eq_subst_scheme_list_eq_free
    simp add: free_tv_subst dom_def)

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

```

lemma *W\_complete*:

```

"[]" |- e :: t' ==> (? S t. (? m. W e [] n = Some(S,t,m)) &
  (? R. t' = $ R t))"
apply (cut_tac A = "[]" and S' = "id_subst" and e = "e" and t' = "t'" in W_complete_lemma)
apply simp_all
done

```

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.