MiniML

Wolfgang Naraschewski and Tobias Nipkow

September 19, 2015

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
\[ \text{option\_bind} :: \text{[\texttt{a option, \texttt{a} => \texttt{b option]} => \texttt{b option}]} \quad \text{where} \]
\[ \text{option\_bind} \ m \ f = (\text{case } m \text{ of } \text{None} => \text{None } | \text{Some } r => f \ r) \]

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)"
\langle proof \rangle

lemma option\_bind\_None: "option\_bind None f = None"
\langle proof \rangle

declare option\_bind\_Some [simp] option\_bind\_None [simp]

— expansion of option\_bind

lemma split\_option\_bind: "P(option\_bind res f) =
\quad ((\text{res} = \text{None } -> P \text{ None}) \ & \ (!s. \text{res} = \text{Some } s \ -> P(f \ s)))"
\langle proof \rangle

lemma option\_bind\_eq\_None [simp]:
\quad "((option\_bind m f) = \text{None}) = (\text{m=\text{None}}) | (\ ? p. \text{m} = \text{Some } p \ & \ f \ p = \text{None})"
\langle proof \rangle
lemma rotate_Some: "(y = Some x) = (Some x = y)"
⟨proof⟩
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
primrec 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

primrec 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)"

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

2
begin

primrec free_tv where
    "free (FVar n) = {n}" \\
| "free (BVar n) = {}" \\
| "free (S1 :=> S2) = (free S1) Un (free S2)"

primrec free_tv_ML where
    "free_ML (FVar n) = [n]" \\
| "free_ML (BVar n) = []" \\
| "free_ML (S1 :=> S2) = (free_ML S1) @ (free_ML S2)"

primrec bound_tv where
    "bound (FVar n) = {}" \\
| "bound (BVar n) = {n}" \\
| "bound (S1 :=> S2) = (bound S1) Un (bound S2)"

primrec min_new_bound where
    "min_new (FVar n) = 0" \\
| "min_new (BVar n) = Suc n" \\
| "min_new (S1 :=> S2) = max (min_new S1) (min_new S2)"

primrec app_subst where
    "$ (FVar n) = mk_scheme (S n)" \\
| "$ (BVar n) = (BVar n)" \\
| "$ (S1 :=> S2) = ($ S1) :=> ($ S2)"

end
"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 (proof)**

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 = (%n. 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 (proof)**

end

**class** typ_of =

**fixes** typ_of :: "'a => typ"

**instantiation** "typ" :: typ_of begin
definition
"typ_of T = T"

instance (proof)
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 (proof)
end

lemma free_tv_subst:
"free_tv S = (dom S) Un (cod S)"
(proof)

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 = $R o U"
and mgu_Some: "$S t1 = $S t2 ==> ? U. mgu t1 t2 = Some U"
and mgu_free: "mgu t1 t2 = Some U ==> (free_tv U) ≤ (free_tv t1) 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)"
(proof)

lemma mk_scheme_injective [rule_format]: "!t'. mk_scheme t = mk_scheme t' =⇒ t=t'"
(proof)

lemma free_tv_mk_scheme: "free_tv (mk_scheme t) = free_tv t"
(proof)

declare free_tv_mk_scheme [simp]

lemma subst_mk_scheme: "$ S (mk_scheme t) = mk_scheme ($ S t)"
(proof)

declare subst_mk_scheme [simp]

— constructor laws for app_subst
lemma app_subst_Nil:
 "S [] = []"
 ⟨proof⟩

lemma app_subst_Cons:
 "S (x#l) = (S x)#(S l)"
 ⟨proof⟩

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)"
 ⟨proof⟩

lemma new_tv_FVar:
 "new_tv n (FVar m) = (m<n)"
 ⟨proof⟩

lemma new_tv_BVar:
 "new_tv n (BVar m) = True"
 ⟨proof⟩

lemma new_tv_Fun:
 "new_tv n (t1 -> t2) = (new_tv n t1 & new_tv n t2)"
 ⟨proof⟩

lemma new_tv_Fun2:
 "new_tv n (t1 =-> t2) = (new_tv n t1 & new_tv n t2)"
 ⟨proof⟩

lemma new_tv_Nil:
 "new_tv n []"
 ⟨proof⟩

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

lemma new_tv_TVar_subst: "new_tv n TVar"
 ⟨proof⟩

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]
lemma new_tv_id_subst [simp]: "new_tv n id_subst"
  ⟨proof⟩

lemma new_if_subst_type_scheme [simp]: "new_tv n (sch::type_scheme) \implies
  \(\lambda k. \text{if } k<n \text{ then } S k \text{ else } S' k\) \ sch = \$S \ sch"
  ⟨proof⟩

lemma new_if_subst_type_scheme_list [simp]: "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"
  ⟨proof⟩

lemma dom_id_subst [simp]: "\text{dom id_subst} = {}"
  ⟨proof⟩

lemma cod_id_subst [simp]: "\text{cod id_subst} = {}"
  ⟨proof⟩

lemma free_tv_id_subst [simp]: "\text{free_tv id_subst} = {}"
  ⟨proof⟩

lemma free_tv_nth_A_impl_free_tv_A [rule_format, simp]:
  "\forall n. n < \text{length } A \implies x : \text{free_tv } (A!n) \implies x : \text{free_tv } A"
  ⟨proof⟩

lemma free_tv_nth_nat_A [rule_format]:
  "\forall nat. nat < \text{length } A \implies x : \text{free_tv } (A!nat) \implies x : \text{free_tv } A"
  ⟨proof⟩

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:
  "\(\forall x:\text{free_tv } t. (S x) = (S' x)) \implies \$S \ (t::typ\) = \$S' \ t"
  ⟨proof⟩

lemma eq_free_eq_subst_te: "\(\forall n. n:(\text{free_tv } t) \implies S1 n = S2 n) \implies \$S1 \ (t::typ) = \$S2 \ t"
  ⟨proof⟩

lemma scheme_substitutions_only_on_free_variables:
  "\(\forall x:\text{free_tv } sch. (S x) = (S' x)) \implies \$S \ (sch::type_scheme\) = \$S' \ sch"
  ⟨proof⟩

lemma eq_free_eq_subst_type_scheme:
  "\(\forall n. n:(\text{free_tv } sch) \implies S1 n = S2 n) \implies \$S1 \ (sch::type_scheme\) = \$S2 \ sch"
  ⟨proof⟩

7
lemma eq_free_eq_subst_scheme_list:
"(!n. n:(free_tv A) --> S1 n = S2 n) ==> $S1 (A::type_scheme list) = $S2 A"
(proof)

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

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"
(proof)

lemma eq_subst_te_eq_free:
"$ S1 (t::typ) = $ S2 t ==> n:(free_tv t) ==> S1 n = S2 n"
(proof)

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

lemma eq_subst_scheme_list_eq_free:
"$S1 (A::type_scheme list) = $S2 A ==> n:(free_tv A) ==> S1 n = S2 n"
(proof)

lemma codD: "v : cod S ==> v : free_tv S"
(proof)

lemma not_free_impl_id: "x ~: free_tv S ==> S x = TVar x"
(proof)

lemma free_tv_le_new_tv: "[| new_tv n t; m:free_tv t |] ==> m<n"
(proof)

lemma cod_app_subst [simp]:
"[| v : free_tv(S n); v~=n |] ==> v : cod S"
(proof)

lemma free_tv_subst_var: "free_tv (S (v::nat)) <= insert v (cod S)"
(proof)

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

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

lemma free_tv_app_subst_scheme_list: "free_tv ($ S (A::type_scheme list)) <= cod S Un free_tv A"
(proof)
lemma free_tv_comp_subst:
"free_tv (%u::nat. $ s1 (s2 u) :: typ) <=
  free_tv s1 Un free_tv s2"
⟨proof⟩

lemma free_tv_o_subst:
"free_tv ($ S1 o S2) <= free_tv S1 Un free_tv (S2 :: nat => typ)"
⟨proof⟩

lemma free_tv_of_substitutionsExtend_to_types:
"n : free_tv t ==> free_tv (S n) <= free_tv ($ S t::typ)"
⟨proof⟩

lemma free_tv_of_substitutionsExtend_to_schemes:
"n : free_tv sch ==> free_tv (S n) <= free_tv ($ S sch::type_scheme)"
⟨proof⟩

lemma free_tv_of_substitutionsExtend_to_scheme_lists [simp]:
"n : free_tv A ==> free_tv (S n) <= free_tv ($ S A::type_scheme list)"
⟨proof⟩

lemma free_tv_ML_scheme:
fixes sch :: type_scheme
shows "(n : free_tv sch) = (n: set (free_tv_ML sch))"
⟨proof⟩

lemma free_tv_ML_scheme_list:
fixes A :: "type_scheme list"
shows "(n : free_tv A) = (n: set (free_tv_ML A))"
⟨proof⟩

lemma bound_tv_mk_scheme [simp]: "bound_tv (mk_scheme t) = {}"
⟨proof⟩

lemma bound_tv_subst_scheme [simp]:
fixes sch :: type_scheme
shows "bound_tv ($ S sch) = bound_tv sch"
⟨proof⟩

lemma bound_tv_subst_scheme_list [simp]:
fixes A :: "type_scheme list"
shows "bound_tv ($ S A) = bound_tv A"
⟨proof⟩

lemma new_tv_subst:
"new_tv n S = ((!m. n <= m --> (S m = TVar m)) &
  (! l. l < n --> new_tv n (S l )))"
proof
lemma new_tv_list: "new_tv n x = (!y:set x. new_tv n y)"
proof
lemma subst_te_new_tv [simp]:
  "new_tv n (t::typ) --> $(%x. if x=n then t' else S x) t = $S t"
proof
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"
proof
lemma subst_tel_new_scheme_list [simp]:
  "new_tv n (A::type_scheme list) = $(%x. if x=n then t else S x) A = $S A"
proof
lemma new_tv_le:
  "n<=m ==> new_tv n t ==> new_tv m t"
proof
lemma [simp]: "new_tv n t = new_tv (Suc n) t"
proof
lemma new_tv_typ_le: "n<=m ==> new_tv n (t::typ) ==> new_tv m t"
proof
lemma new_scheme_list_le: "n<=m ==> new_tv n (A::type_scheme list) ==> new_tv m A"
proof
lemma new_tv_subst_le: "n<=m ==> new_tv n (S::subst) ==> new_tv m S"
proof
lemma new_tv_subst_var:
  "[| n<m; new_tv m (S::subst) |] ==> new_tv m (S n)"
proof
lemma new_tv_subst_te [simp]:
  "new_tv n S --> new_tv n (t::typ) --> new_tv n ($ S t)"
proof
lemma new_tv_subst_type_scheme [rule_format, simp]:
  "new_tv n S --> new_tv n (sch::type_scheme) --> new_tv n ($ S sch)"
proof
lemma new_tv_subst_scheme_list [simp]:
  "new_tv n S --> new_tv n (A::type_scheme list) --> new_tv n ($ S A)"
proof
lemma new_tv_Suc_list: "new_tv n A --> new_tv (Suc n) ((TVar n)#A)"
proof

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'"
⟨proof⟩

lemma new_tv_only_depends_on_free_tv_scheme_list:
fixes A :: "type_scheme list"
sows "free_tv A = free_tv A' =⇒ new_tv n A =⇒ new_tv n A'"
⟨proof⟩

lemma new_tv_nth_nat_A [rule_format]:
"!nat. nat < length A --> new_tv n A --> (new_tv n (A!nat))"
⟨proof⟩

lemma new_tv_subst_comp_1 [simp]:
"[| new_tv n (S::subst); new_tv n R |] =⇒ new_tv n (($ R) o S)"
⟨proof⟩

lemma new_tv_subst_comp_2 [simp]:
"[| new_tv n (S::subst); new_tv n R |] =⇒ new_tv n (λv.$ R (S v))"
⟨proof⟩

lemma new_tv_not_free_tv [simp]:
"new_tv n A =⇒ n":(free_tv A)"
⟨proof⟩

lemma fresh_variable_types [simp]: "!!t::typ. ? n. (new_tv n t)"
⟨proof⟩

lemma fresh_variable_type_schemes [simp]:
"!!sch::type_scheme. ? n. (new_tv n sch)"
⟨proof⟩

lemma fresh_variable_type_scheme_lists [simp]:
"!!A::type_scheme list. ? n. (new_tv n A)"
⟨proof⟩

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)"
⟨proof⟩

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')"
⟨proof⟩

lemma mgu_new:
"[| mgu t1 t2 = Some u; new_tv n t1; new_tv n t2 |] =⇒ new_tv n u"
⟨proof⟩
lemma length_app_subst_list [simp]:
  "!!A:: ('a::type_struct) list. length ($ S A) = length A"
⟨proof⟩

lemma subst_TVar_scheme [simp]:
  fixes sch :: type_scheme
  shows "$ TVar sch = sch"
⟨proof⟩

lemma subst_TVar_scheme_list [simp]:
  fixes A :: "type_scheme list"
  shows "$ TVar A = A"
⟨proof⟩

lemma app_subst_id_te [simp]: "$ id_subst = (%t::typ. t)"
⟨proof⟩

lemma app_subst_id_type_scheme [simp]:
  "$ id_subst = (%sch::type_scheme. sch)"
⟨proof⟩

lemma app_subst_id_tel [simp]:
  "$ id_subst = (%A::type_scheme list. A)"
⟨proof⟩

lemma id_subst_sch [simp]:
  fixes sch :: type_scheme
  shows "$ id_subst sch = sch"
⟨proof⟩

lemma id_subst_A [simp]:
  fixes A :: "type_scheme list"
  shows "$ id_subst A = A"
⟨proof⟩

lemma o_id_subst [simp]: "$S o id_subst = S"
⟨proof⟩

lemma subst_comp_te: "$ R ($ S t::typ) = $ (%x. $ R (S x) ) t"
⟨proof⟩

lemma subst_comp_type_scheme:
  "$ R ($ S sch::type_scheme) = $ (%x. $ R (S x) ) sch"
⟨proof⟩

lemma subst_comp_scheme_list:
  "$ R ($ S A::type_scheme list) = $ (%x. $ R (S x)) A"
⟨proof⟩

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"
⟨proof⟩

lemma subst_id_on_type_scheme_list:
  fixes A :: "type_scheme list"
  shows "!x : free_tv A. S x = (TVar x) ==> \$ S A = A"
⟨proof⟩

lemma nth_subst [rule_format]:
  "!n. n < length A --> ($ S A)!n = ($S (A!n))"
⟨proof⟩

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 = (? S. t = (bound_typ_inst S sch))"

instantiation type_scheme :: ord
begin

definition le_type_scheme_def: "sch' <= (sch::type_scheme) <-> (!t. t <| sch' --> t <| sch)"

definition
  "(sch' < (sch :: type_scheme)) <-> sch' \leq sch \land sch' \neq sch"

instance ⟨proof⟩

end

primrec subst_to_scheme :: "[nat => type_scheme, typ] => type_scheme" where
  "subst_to_scheme B (TVar n) = (B n)"
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 ⟨proof⟩end

lemmas for instatiation

lemma bound_typ_inst_mk_scheme [simp]: "bound_typ_inst S (mk_scheme t) = t" ⟨proof⟩

lemma bound_typ_inst_composed_subst [simp]:
"bound_typ_inst ($S o R) ($S sch) = $S (bound_typ_inst R sch)"
⟨proof⟩

lemma bound_typ_inst_eq:
"S = S' ==> sch = sch' ==> bound_typ_inst S sch = bound_typ_inst S' sch'"
⟨proof⟩

lemma bound_scheme_inst_mk_scheme [simp]:
"bound_scheme_inst B (mk_scheme t) = mk_scheme t" ⟨proof⟩

lemma substitution_lemma: "$S (bound_scheme_inst B sch) = (bound_scheme_inst ($S o B) ($ S sch))"
⟨proof⟩

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))"
⟨proof⟩

lemma subst_to_scheme_inverse:
"new_tv n sch ⇒
 subst_to_scheme (%k. if n <= k then BVar (k - n) else FVar k)
 (bound_typ_inst (%k. TVar (k + n)) sch) = sch"
⟨proof⟩

lemma aux: "t = t' ==> 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'"
proof

lemma aux2: "new_tv n sch ⟹ 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"
≤proof≤

lemma le_type_scheme_def2:
  fixes sch sch' :: type_scheme
  shows "(sch' <= sch) = (? B. sch' = bound_scheme_inst B sch)"
≤proof≤

lemma le_type_eq_is_bound_typ_instance [rule_format]: "(mk_scheme t) <= sch = t <| sch"
≤proof≤

lemma le_env_Cons [iff]:
  "(sch # A <= sch' # B) = (sch <= (sch':type_scheme) & A <= B)"
≤proof≤

lemma is_bound_typ_instance_closed_subst: "t <| sch ==> $S t <| $S sch"
≤proof≤

lemma S_compatible_le_scheme:
  fixes sch sch' :: type_scheme
  shows "sch' <= sch ==> $S sch' <= $ S sch"
≤proof≤

lemma S_compatible_le_scheme_lists:
  fixes A A' :: "type_scheme list"
  shows "A' <= A ==> $S A' <= $ S A"
≤proof≤

lemma bound_typ_instance_trans: "[| t <| sch; sch <= sch' |] ==> t <| sch'"
≤proof≤

lemma le_type_scheme_refl [iff]: "sch <= (sch::type_scheme)"
≤proof≤

lemma le_env_refl [iff]: "A <= (A::type_scheme list)"
≤proof≤

lemma bound_typ_instance_BVar [iff]: "sch <= BVar n"
≤proof≤

lemma le_FVar [simp]: "(sch <= FVar n) = (sch = FVar n)"
≤proof≤
lemma not_FVar_le_Fun [iff]: "¬(FVar n <= sch1 =⇒ sch2)"
⟨proof⟩

lemma not_BVar_le_Fun [iff]: "¬(BVar n <= sch1 =⇒ sch2)"
⟨proof⟩

lemma Fun_le_FunD:
"(sch1 =⇒ sch2 <= sch1' =⇒ sch2') ==> sch1 <= sch1' & sch2 <= sch2'"
⟨proof⟩

lemma scheme_le_Fun: "(sch' <= sch1 =⇒ sch2) =⇒ ? sch'1 sch'2. sch' = sch'1 =⇒ sch'2"
⟨proof⟩

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"
⟨proof⟩

lemma gen_without_effect [simp]:
"(free_tv t) <= (free_tv sch) ==> gen sch t = (mk_scheme t)"
⟨proof⟩

lemma free_tv_gen [simp]:
"free_tv (gen ($ S A) t) = free_tv t Int free_tv ($ S A)"
⟨proof⟩

lemma free_tv_gen_cons [simp]:
"free_tv (gen ($ S A) t # $ S A) = free_tv ($ S A)"
⟨proof⟩

lemma bound_tv_gen [simp]:
"bound_tv (gen A t1) = (free_tv t1) - (free_tv A)"
⟨proof⟩

lemma new_tv_compatible_gen: "new_tv n t ==> new_tv n (gen A t)"
⟨proof⟩

lemma gen_eq_gen_ML: "gen A t = gen_ML A t"
⟨proof⟩

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)"
⟨proof⟩

lemma bound_typ_inst_gen [simp]:
"free_tv(t::typ) <= free_tv(A) == bound_typ_inst S (gen A t) = t"
⟨proof⟩

lemma gen_bound_typ_instance:
"gen ($ S A) ($ S t) <= $ S (gen A t)"
⟨proof⟩

lemma free_tv_subset_gen_le:
"free_tv B <= free_tv A ==> gen A t <= gen B t"
⟨proof⟩

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)"
⟨proof⟩

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"
    ("((_) |-/ (_) :: (_))"

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"
⟨ proof ⟩
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"
⟨ proof ⟩
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"
⟨ proof ⟩

lemma alpha_A':
  "!!A::type_scheme list. $(%x. TVar (if x : free_tv A then x else n + x)) A = $ id_subst A"
⟨ proof ⟩
lemma alpha_A:
"!!A::type_scheme list. $ (\%x. TVar (if x : free_tv A then x else n + x)) A = A"
⟨proof⟩

lemma S_o_alpha_typ:
"$ (S o alpha) (t::typ) = S ($ (\%x. TVar (alpha x)) t)"
⟨proof⟩

lemma S_o_alpha_typ':
"$ (\%u. (S o alpha) u) (t::typ) = S ($ (\%x. TVar (alpha x)) t)"
⟨proof⟩

lemma S_o_alpha_type_scheme:
"$ (S o alpha) (sch::type_scheme) = S ($ (\%x. TVar (alpha x)) sch)"
⟨proof⟩

lemma S_o_alpha_type_scheme_list:
"$ (S o alpha) (A::type_scheme list) = S ($ (\%x. TVar (alpha x)) A)"
⟨proof⟩

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"
⟨proof⟩

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"
⟨proof⟩

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)"
⟨proof⟩

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)"
⟨proof⟩

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}"
⟨proof⟩
lemma new_tv_Int_free_tv_empty_type: "!!t::typ. new_tv n t ==> {x. ? y. x = n + y} Int free_tv t = {}"
⟨proof⟩
lemma new_tv_Int_free_tv_empty_scheme: "!!sch::type_scheme. new_tv n sch ==> {x. ? y. x = n + y} Int free_tv sch = {}"
⟨proof⟩
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 = {}"
⟨proof⟩

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)"
⟨proof⟩

declare has_type.intros [intro!]

lemma has_type_le_env [rule_format (no_asm)]: "A |- e::t ==> !B. A <= B --> B |- e::t"
⟨proof⟩
lemma has_type_cl_sub: "A |- e :: t ==> !S. $S A |- e :: $S t"
⟨proof⟩

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
primrec W :: "[expr, ctxt, nat] => result_W" where
 "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 ) )"
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 \(W_{\text{var-ge}}\) [rule_format (no_asm)]:
  "!A n S t m. W e A n = Some (S,t,m) --> n<=m"
(proof)

declare \(W_{\text{var-ge}}\) [simp]

lemma \(W_{\text{var-ge}}\D:
  "Some (S,t,m) = W e A n ==> n<=m"
(proof)

lemma new_tv_compatible_W:
  "new_tv n A ==> Some (S,t,m) = W e A n ==> new_tv m A"
(proof)

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)"
(proof)

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"
(proof)

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)
⟨proof⟩
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"
⟨proof⟩

lemma weaken_A_Int_B_eq_empty: "(!x. x : A --> x ^: B) ==> A Int B = {}"
⟨proof⟩

lemma weaken_not_elem_A_minus_B: "x ^: A | x : B ==> x ^: A - B"
⟨proof⟩

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"
⟨proof⟩

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))"
⟨proof⟩

lemma W_complete:
"[] |- e :: t' ==> (? S t. (? m. W e [] n = Some(S,t,m)) &
  (? R. t' = $ R t))"
⟨proof⟩

end

References

In E. Giménez and C. Paulin-Mohring, editors, Types for Proofs and Programs: Intl.