MiniML

Wolfgang Naraschewski and Tobias Nipkow

March 17, 2025

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)
syntax_consts "_option_bind" == option_bind
translations "P := E; F" == "CONST option_bind E (λP. 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 split_option_bind_asm: "P(option_bind res f) =
```

```
        (~ ((res = None ∧ ¬ P None) ∨ (∃s. res = Some s ∧ ¬ 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)

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) ⟹
    $(λk. if k<n then S k else S' k) sch = $S sch"
  by (induction sch) simp_all

lemma new_if_subst_type_scheme_list: "new_tv n (A::type_scheme list) ⟹
    $(λk. if k<n then S k 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:
  "⟦ n < length A;  x : free_tv (A!n) ⟧ ⟹ x : 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:
  "(∀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"
  using typ_substitutions_only_on_free_variables
  by force

lemma scheme_substitutions_only_on_free_variables:
  "(⋀x. x ∈ free_tv sch ⟹ S x = S' x) ⟹ $ S (sch::type_scheme) = $ S' sch"
  by (induct sch) simp_all
```

```
lemma eq_free_eq_subst_scheme_list:
  "(⋀n. n ∈ free_tv A ⟹ S1 n = S2 n) ⟹ $S1 (A::type_scheme list) = $S2 A"
by (induct A) (fastforce intro: scheme_substitutions_only_on_free_variables)+

lemma weaken_asm_Un: "((∀x∈A. (P x)) ⟶ Q) ⟹ ((∀x∈A ∪ B. (P x)) ⟶ Q)"
  by fast

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:
  "⟦ $ S1 (sch::type_scheme) = $ S2 sch; n:(free_tv sch) ⟧ ⟹ 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) ⟧ ⟹ 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 ⟹ 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:
  "⟦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)"
  using cod_app_subst by blast

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 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 ∘ S2) ⊆ free_tv S1 ∪ 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 ⟹ 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 ⟹ 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 ⟹ 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. S m = TVar m) ∧ (∀l<n. new_tv n (S l)))"
proof
  assume *"new_tv n S"*
  then show *"(∀m≥n. S m = TVar m) ∧ (∀l<n. new_tv n (S l))"*
    by *(metis codD cod_app_subst linorder_not_less new_tv_def*
      not_free_impl_id)
next
  assume *"(∀m≥n. S m = TVar m) ∧ (∀l<n. new_tv n (S l))"*
  with *linorder_not_less* **show** *"new_tv n S"*
    unfolding *new_tv_def free_tv_subst cod_def dom_def*
    by *blast*
qed

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"*
using *Suc_n_not_le_n nat_le_linear new_tv_le* **by** *blast*

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' ⟹ new_tv n A ⟹ new_tv n A'"*
unfolding *new_tv_def* **by** *simp*

lemma *new_tv_nth_nat_A:*
"m < length A ⟹ new_tv n A ⟹ 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 |] ==> new_tv n (($ R) ∘ S)"
by *(simp add: new_tv_subst)*

lemma *new_tv_subst_comp_2 :*
"[| new_tv n (S::subst); new_tv n R |] ==> new_tv n (λv.$ 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 ⟹ n ∉ free_tv A"
  using free_tv_le_new_tv less_irrefl_nat by blast

lemma fresh_variable_types: "∃n. new_tv n (t::typ)"
unfolding new_tv_def
proof (induction t)
  case (Fun t1 t2)
  then show ?case
    by (metis Type.free_tv_Fun UnE dual_order.strict_trans linorder_neq_iff)
qed auto

lemma fresh_variable_type_schemes:
  "∃n. new_tv n (sch::type_scheme)"
unfolding new_tv_def
proof (induction sch)
  case (SFun sch1 sch2)
  then show ?case
    by (meson List.finite_set finite_nat_set_iff_bounded free_tv_ML_scheme)
qed auto

lemma fresh_variable_type_scheme_lists:
  "∃n. new_tv n (A::type_scheme list)"
proof (induction A)
  case Nil
  then show ?case by auto
next
  case (Cons a A)
  then show ?case
    by (metis fresh_variable_type_schemes le_cases new_tv_Cons new_tv_le)
qed

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"
  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 ∘ 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) = $ (λx. $ R (S x) ) sch"
  by (induct sch) simp_all

lemma subst_comp_scheme_list:
  "$ R ($ S A::type_scheme list) = $ (λx. $ R (S x)) A"
  unfolding app_subst_list
  by (induct A) (auto simp add: subst_comp_type_scheme)

lemma nth_subst:
  "n < length A ⟹ ($ 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 = (∃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' ≤ sch ∧ sch' ≠ 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 instatiation

```
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:
    "mk_scheme t = bound_scheme_inst B sch ⟹
          (∃S. ∀x∈bound_tv sch. B x = mk_scheme (S x))"
proof (induction "sch" arbitrary: t)
  case (BVar x)
  then show ?case
    by (force intro: sym)
next
  case (SFun type_scheme1 type_scheme2 t)
  obtain S1 where S1: "∀x∈bound_tv type_scheme1. B x = mk_scheme (S1 x)"
    by (metis SFun.IH(1) SFun.prems bound_scheme_inst.simps(3) mk_scheme_Fun)
  obtain S2 where S2: "∀x∈bound_tv type_scheme2. B x = mk_scheme (S2 x)"
    by (metis SFun.IH(2) SFun.prems bound_scheme_inst.simps(3) mk_scheme_Fun)
  let ?S = "λx. if x:bound_tv type_scheme1 then (S1 x) else (S2 x)"
  show ?case
  proof
    show "∀x∈bound_tv (type_scheme1 =-> type_scheme2). B x = mk_scheme (?S x)"
      using S1 S2 by auto
  qed
qed auto

lemma subst_to_scheme_inverse:
  "new_tv n sch ⟹
    subst_to_scheme (λk. if n ≤ k then BVar (k - n) else FVar k)
      (bound_typ_inst (λk. TVar (k + n)) sch) = sch"
by (induction sch) auto

lemma aux: "t = t' ⟹
      subst_to_scheme (λk. if n ≤ k then BVar (k - n) else FVar k) t =
```

```
    subst_to_scheme (λk. if n ≤ k then BVar (k - n) else FVar k) t'"
  by blast

lemma aux2: "new_tv n sch ⟹
  subst_to_scheme (λk. if n ≤ k then BVar (k - n) else FVar k) (bound_typ_inst S sch)
=
    bound_scheme_inst ((subst_to_scheme (λk. if n ≤ k then BVar (k - n) else FVar k))
∘ 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)"
proof -
  have *: "bound_typ_inst S (bound_scheme_inst B sch) =
          bound_typ_inst (λn. bound_typ_inst S (B n)) sch" for S B
    by (induction sch) auto
  show ?thesis
    by (metis (no_types, lifting) "*" aux2 fresh_variable_type_schemes
        is_bound_typ_instance le_type_scheme_def new_tv_Fun2 subst_to_scheme_inverse)
qed

lemma le_type_eq_is_bound_typ_instance: "(mk_scheme t) ≤ sch = t <| sch"
  using bound_typ_inst_mk_scheme is_bound_typ_instance le_type_scheme_def by presburger

lemma le_env_Cons [iff]:
  "(sch # A ≤ sch' # B) = (sch ≤ (sch'::type_scheme) ∧ A ≤ B)"
proof (intro iffI)
  assume "sch # A ≤ sch' # B" then show "sch ≤ sch' ∧ A ≤ B"
    by (smt (verit) Suc_length_conv Suc_mono le_env_def nat.inject nth_Cons_0
      nth_Cons_Suc zero_less_Suc)
next
  assume "sch ≤ sch' ∧ A ≤ B" then show "sch # A ≤ sch' # B"
    by (smt (verit, ccfv_SIG) le_env_def length_Cons less_Suc_eq_0_disj nth_Cons_0
        nth_Cons_Suc)
qed

lemma is_bound_typ_instance_closed_subst: "t <| sch ⟹ $S t <| $S sch"
  by (metis bound_typ_inst_composed_subst is_bound_typ_instance)

lemma S_compatible_le_scheme:
  fixes sch sch' :: type_scheme
  shows "sch' ≤ sch ⟹ $S sch' ≤ $ S sch"
  using le_type_scheme_def2 substitution_lemma
  by force

lemma S_compatible_le_scheme_lists:
  fixes A A' :: "type_scheme list"
  shows "A' ≤ A ⟹ $S A' ≤ $ S A"
```

```
  by (simp add: S_compatible_le_scheme le_env_def nth_subst)

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"
  using le_type_scheme_def2 by auto

lemma le_FVar [simp]: "(sch ≤ FVar n) = (sch = FVar n)"
  by (simp add: le_type_scheme_def2)

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)"
  by (simp add: le_type_scheme_def2)

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:
  fixes sch'::type_scheme
  shows "sch ≤ sch' ⟹ free_tv sch' ≤ free_tv sch"
proof (induction "sch" arbitrary: sch')
  case (FVar x)
  then show ?case
    by (induction "sch'") auto
next
  case (BVar x)
  then show ?case
    by (induction "sch'") auto
next
  case (SFun sch1 sch2)
  then show ?case
  proof (induction sch')
    case (SFun sch'1 sch'2)
    then show ?case
      by (metis Fun_le_FunD Un_mono free_tv_type_scheme.simps(3))
  qed auto
```

```
qed

lemma le_env_free_tv:
  fixes A :: "type_scheme list"
  assumes "A ≤ B"
  shows "free_tv B ≤ free_tv A"
  using assms
proof (induction B arbitrary: A)
  case Nil
  then show ?case
    by auto
next
  case (Cons b B)
  then obtain a A' where "A = a # A'" "a ≤ b" "A' ≤ B"
    by (metis le_env_Cons le_env_def length_0_conv neq_Nil_conv)
  with Cons show ?case
    using le_type_scheme_free_tv by fastforce
qed

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"
  unfolding gen_ML_def
  by (induct t) (auto simp: free_tv_ML_scheme_list)

lemma gen_subst_commutes:
  "free_tv S ∩ (free_tv t - free_tv A) = {} ⟹ gen ($ S A) ($ S t) = $ S (gen A t)"
proof (induct t)
  case (TVar x)
  show ?case
  proof (cases "x ∈ free_tv A")
    case True
    then show ?thesis
      by simp
  next
    case False
    then have "x ∉ free_tv S"
      using TVar Type.free_tv_TVar by blast
    then show ?thesis
      using False free_tv_app_subst_scheme_list free_tv_subst not_free_impl_id
      by fastforce
  qed
next
  case (Fun t1 t2)
  then show ?case
    by (simp add: Diff_eq Int_Un_distrib2 disjoint_iff)
qed

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

```
  have "bound_typ_inst R (gen ($ S A) ($ S t)) =
       bound_typ_inst (λa. bound_typ_inst R (gen ($ S A) (S a)))
        ($ S (gen A t))" for R
    by (induction t) auto
  then show ?thesis
    using is_bound_typ_instance le_type_scheme_def by auto
qed

lemma free_tv_subset_gen_le:
  assumes "free_tv B ⊆ free_tv A"
  shows "gen A t ≤ gen B t"
proof -
  have "bound_typ_inst S (gen A t) =
       bound_typ_inst (λb. if b ∈ free_tv A then TVar b else S b) (gen B t)" for S
    using assms
    by (induction t) force+
  then show ?thesis
    using is_bound_typ_instance le_type_scheme_def by auto
qed

lemma gen_t_le_gen_alpha_t [simp]:
  assumes "new_tv n A"
  shows "gen A t ≤ gen A ($ (λx. TVar (if x ∈ free_tv A then x else n + x)) t)"
proof -
  have "bound_typ_inst S (gen A t) =
       bound_typ_inst (λx. S (if n ≤ x then x - n else x))
        (gen A ($ (λx. TVar (if x ∈ free_tv A then x else n + x)) t))" for S
  proof (induction t)
    case (TVar x)
    then show ?case
      using assms free_tv_le_new_tv by auto
  next
    case (Fun t1 t2)
    then show ?case
      by force
  qed
  then show ?thesis
    using is_bound_typ_instance le_type_scheme_def by auto
qed

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[simp]:`
```
  "⋀t::typ. $(λn. if n : (free_tv A) Un (free_tv t) then (S n) else (TVar n)) t = $S
t"
```
  using `UnCI eq_free_eq_subst_te` **by** `fastforce`

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"
```
  using `eq_free_eq_subst_scheme_list` **by** `fastforce`

lemma `replace_s_by_s':`
```
   "$(λn. if n ∈ free_tv A ∪ free_tv t then S n else TVar n) A
     ⊢ e :: $(λn. if n ∈ free_tv A ∪ free_tv t then S n else TVar n) t
  ⟹ $S A ⊢ e :: $S t"
```
  by `(metis (mono_tags, lifting) s'_a_equals_s_a s'_t_equals_s_t)`

lemma `alpha_A':`
```
  "⋀A::type_scheme list. $ (λx. TVar (if x : free_tv A then x else n + x)) A = $ id_subst
A"
```
  by `(simp add: eq_free_eq_subst_scheme_list id_subst_def)`

lemma `alpha_A:`
```
  "⋀A::type_scheme list. $ (λx. TVar (if x : free_tv A then x else n + x)) A = A"
```
  by `(simp add: alpha_A')`

lemma `S_o_alpha_typ:`
```
  "$ (S ∘ alpha) (t::typ) = $ S ($ (λx. TVar (alpha x)) t)"
```
  by `(induct_tac "t") auto`

lemma `S_o_alpha_type_scheme:`
```
  "$ (S ∘ alpha) (sch::type_scheme) = $ S ($ (λx. TVar (alpha x)) sch)"
```

```
  by (induct_tac "sch") auto

lemma S_o_alpha_type_scheme_list:
  "$ (S ∘ alpha) (A::type_scheme list) = $ S ($ (λx. TVar (alpha x)) A)"
proof (induction "A")
  case Nil
  then show ?case by auto
next
  case (Cons a A)
  then show ?case
    by (metis S_o_alpha_type_scheme app_subst_Cons)
qed

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) ∘
         (λx. if x : free_tv A then x else n + x)) A"
  using eq_free_eq_subst_scheme_list by fastforce

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"
  using Type.dom_def by auto

lemma cod_S':
  "⋀(A::type_scheme list) (t::typ).
   cod (λn. if n : free_tv A Un free_tv t then S n else TVar n) ⊆
   free_tv ($ S A) Un free_tv ($ S t)"
  unfolding free_tv_subst cod_def subset_eq Type.dom_def
  by (smt (verit, del_insts)  UN_iff Un_iff
      free_tv_of_substitutions_extend_to_scheme_lists
      free_tv_of_substitutions_extend_to_types mem_Collect_eq subsetD)

lemma free_tv_S':
 "⋀(A::type_scheme list) (t::typ).
  free_tv (λn. 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)"
  unfolding free_tv_subst
  using dom_S' cod_S' by blast

lemma free_tv_alpha:
  fixes t1::"typ"
  shows "(free_tv ($ (λx. TVar (if x : free_tv A then x else n + x)) t1) - free_tv A)
⊆
          {x. ∃y. x = n + y}"
  by (induction t1) auto

lemma new_tv_Int_free_tv_empty_type: "new_tv n t ⟹ {x. ∃y. x = n + y} ∩ free_tv t
= {}"
```

```
  using free_tv_le_new_tv by fastforce

lemma new_tv_Int_free_tv_empty_scheme_list:
  fixes A :: "type_scheme list"
  shows "new_tv n A ⟹ {x. ∃y. x = n + y} ∩ free_tv A = {}"
proof (induction A)
  case Nil
  then show ?case
    by auto
next
  case (Cons a A)
  then show ?case
    using new_tv_Int_free_tv_empty_type by blast
qed

declare has_type.intros [intro!]

lemma has_type_le_env: "A ⊢ e::t ⟹ A ≤ B ⟹ B ⊢ e::t"
proof (induction arbitrary: B rule: has_type.induct)
  case (VarI n A t)
  then show ?case
    by (simp add: le_env_def le_type_scheme_def)
next
  case (LETI A e1 t1 e2 t)
  then show ?case
    by (meson free_tv_subset_gen_le has_type.LETI le_env_Cons le_env_free_tv)
qed auto

— has_type is closed w.r.t. substitution
lemma has_type_cl_sub: "A ⊢ e :: t ⟹ $S A ⊢ e :: $S t"
proof (induction arbitrary: S rule: has_type.induct)
  case (LETI A e1 t1 e2 t)
  obtain n where n: "new_tv n ($ S A)" "new_tv n A" "new_tv n t"
                  "new_tv n ($ S t)"
    using ex_fresh_variable by blast
  define F where "F ≡ λi. if i ∈ free_tv A ∪ free_tv t then S i else TVar i"
  define G where "G ≡ λi. if i ∈ free_tv A then i else n + i"
  have 1: "$ (F ∘ G) A ⊢ e1 :: $ (F ∘ G) t1"
    by (simp add: LETI.IH)
  have "free_tv F ⊆ free_tv A Un free_tv ($ S A) Un free_tv t Un free_tv ($ S t)"
    using F_def free_tv_S' by presburger
  moreover
  have "(free_tv ($ (λi. TVar (G i)) t1) - free_tv A) ⊆ {x. ∃y. x = n + y}"
    by (simp add: G_def free_tv_alpha)
  ultimately
  have "free_tv F ∩ (free_tv ($ (λi. TVar (G i)) t1) - free_tv A) = {}"
    using not_add_less1 n by (fastforce simp: new_tv_def)
  moreover
  have "gen A t ≤ gen A ($ (λi. TVar (G i)) t)" for t
```

```
    using n(2) by (auto simp: G_def)
  then have "$ F (gen A ($ (λi. TVar (G i)) t1)) # $ F A ⊢ e2 :: $ F t"
    using LETI.IH(2) S_compatible_le_scheme has_type_le_env by fastforce
  ultimately have "$ F A ⊢ LET e1 e2 :: $ F t"
    by (metis (mono_tags, lifting) "1" G_def has_type.LETI S_o_alpha_typ
        comp_apply eq_free_eq_subst_scheme_list gen_subst_commutes)
  then show ?case
    by (metis F_def Un_iff eq_free_eq_subst_scheme_list typ_substitutions_only_on_free_variables)
qed (auto simp: nth_subst)

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 ∘ $S2 ∘ 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 ∘ S1, t2, m2) )"


declare Suc_le_lessD [simp]

inductive_simps has_type_simps:
  "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:
  "W e A n = Some (S,t,m) ⟹
        (v∈free_tv S ∨ v∈free_tv t) ⟹ v<n ⟹ v∈free_tv A"
proof (induction e arbitrary: n A S t m v)
  case (Var i)
  show ?case
  proof (cases "v ∈ free_tv (A!i)")
    case True
    with Var show ?thesis
      by (metis W.simps(1) free_tv_nth_A_impl_free_tv_A not_None_eq)
  next
    case False
    with Var show ?thesis
      by (force simp: o_def free_tv_nth_A_impl_free_tv_A dest: free_tv_bound_typ_inst1

          split: if_split_asm)
  qed
next
  case (Abs e n A S t m v)
  then obtain S1 t1 n1 where "W e (FVar n # A) (Suc n) = Some (S1, t1, n1)"
    by (metis (lifting) W.simps(2) not_None_eq option_bind_eq_None prod_cases3)
  then show ?case
    using Abs.IH [of "FVar n # A" "Suc n" S1 t1 n1 v] Abs.prems
    by (force simp: codD cod_app_subst)
next
  case (App e1 e2 n A S t m v)
  then show ?case
  proof (clarsimp split: split_option_bind_asm prod.split_asm)
    fix S' t' n1 S1 t1 n2 S2
    assume v: "v ∈ free_tv ($ S2 ∘ $ S1 ∘ S') ∨ v ∈ free_tv (S2 n2)"
      and "v < n"
      and e1: "W e1 A n = Some (S', t', n1)"
      and e2: "W e2 ($ S' A) n1 = Some (S1, t1, n2)"
```

```
      and mgu: "mgu ($ S1 t') (t1 -> TVar n2) = Some S2"
    have n: "n ≤ n1" "n1 ≤ n2"
      using e1 e2 by auto
    show "v ∈ free_tv A"
      using v
    proof
      assume v1: "v ∈ free_tv ($ S2 ∘ $ S1 ∘ S')"
      then have "v ∈ free_tv S2 ∪ free_tv (λx. $ S1 (S' x))"
        by (metis (no_types, lifting) ext comp_apply free_tv_o_subst fun.map_comp
            subsetD)
      moreover
      have "free_tv S2 ⊆ insert n2 (free_tv ($ S1 t') ∪ free_tv t1)"
        using mgu mgu_free by fastforce
      ultimately
      show "v ∈ free_tv A"
        using App.IH n v1 ‹v<n› e1 e2 codD free_tv_app_subst_scheme_list
        by (smt (verit, ccfv_threshold) Un_iff free_tv_app_subst_te free_tv_o_subst
            fun.map_comp insert_iff linorder_not_less order.strict_trans2 subsetD)
    next
      assume v2: "v ∈ free_tv (S2 n2)"
      then have "v < n1"
        using App.prems n by linarith
      then have "free_tv S2 ⊆ free_tv ($ S1 t') ∪ free_tv (t1 -> TVar n2)"
        using mgu mgu_free by blast
      then show "v ∈ free_tv A"
        using App.IH n v2 ‹v<n› ‹v<n1› e1 e2 codD free_tv_app_subst_scheme_list
        by (smt (verit, ccfv_threshold) UnE  cod_app_subst empty_iff
            free_tv_app_subst_te free_tv_typ.simps insert_iff linorder_not_less subsetD)
    qed
  qed
next
  case (LET e1 e2 n A S t2 n3 v)
  then show ?case
  proof (clarsimp split: split_option_bind_asm prod.split_asm)
    fix S1 t1 n2 S2
    assume "v ∈ free_tv ($ S2 ∘ S1) ∨ v ∈ free_tv t2"
      and "v < n"
      and "W e1 A n = Some (S1, t1, n2)"
      and "W e2 (gen ($ S1 A) t1 # $ S1 A) n2 = Some (S2, t2, n3)"
    with LET.IH
    show "v ∈ free_tv A"
      by (smt (verit) Un_iff W_var_geD codD free_tv_app_subst_scheme_list
          free_tv_gen_cons free_tv_o_subst order.strict_trans2 subsetD)
  qed
qed

lemma weaken_A_Int_B_eq_empty: "(∀x. x ∈ A ⟶ x ∉ B) ⟹ A ∩ B = {}"
  by blast
```

```
lemma weaken_not_elem_A_minus_B: "x ∉ A ∨ x ∈ B ⟹ x ∉ A - B"
  by blast
```

— correctness of W with respect to *has_type*

```
lemma W_correct_lemma: "⟦new_tv n A; Some (S,t,m) = W e A n⟧ ⟹ $S A ⊢ e :: t"
proof (induction "e" arbitrary: A S t m n)
  case Var thus ?case
    using is_bound_typ_instance by (auto split: if_splits)
next
  case (Abs e) thus ?case
    apply (simp split: split_option_bind_asm prod.splits)
    by (metis AbsI app_subst_Cons app_subst_type_scheme.simps(1) lessI new_tv_Cons
        new_tv_FVar new_tv_Suc)
next
  case (App e1 e2)
  then show ?case
  proof (simp split: split_option_bind_asm prod.splits)
    fix S1 t1 n1 S2 t2 n2 S3
    assume e1: "W e1 A n = Some (S1, t1, n1)"
      and e2: "W e2 ($ S1 A) n1 = Some (S2, t2, n2)"
      and mgu: "mgu ($ S2 t1) (t2 -> TVar n2) = Some S3"
    show "$ (λa. $ S3 ($ S2 (S1 a))) A ⊢ App e1 e2 :: S3 n2"
    proof (rule has_type.AppI)
      have "$ S3 (t2 -> TVar n2) = $ S3 ($ S2 t1)"
        using mgu mgu_eq by presburger
      with App show "$ (λa. $ S3 ($ S2 (S1 a))) A ⊢ e1 :: $ S3 t2 -> S3 n2"
        by (metis (no_types) Type.app_subst_Fun Type.app_subst_TVar e1 has_type_cl_sub
subst_comp_scheme_list)
      show "$ (λa. $ S3 ($ S2 (S1 a))) A ⊢ e2 :: $ S3 t2"
        using e1 e2 mgu App
        by (metis has_type_cl_sub new_tv_W new_tv_compatible_W new_tv_subst_scheme_list
            subst_comp_scheme_list)
    qed
  qed
next
  case (LET e1 e2) thus ?case
  proof (simp split: split_option_bind_asm prod.splits)
    fix S1 t1 m1 S2
    assume "new_tv n A"
      and e1: "W e1 A n = Some (S1, t1, m1)"
      and e2: "W e2 (gen ($ S1 A) t1 # $ S1 A) m1 = Some (S2, t, m)"
    show "$ (λa. $ S2 (S1 a)) A ⊢ LET e1 e2 :: t"
    proof (rule has_type.LETI)
      show "$ (λa. $ S2 (S1 a)) A ⊢ e1 :: $ S2 t1"
        using LET e1 by (metis (no_types, lifting) has_type_cl_sub subst_comp_scheme_list)
      have "free_tv S2 ∩ (free_tv t1 - free_tv ($ S1 A)) = {}"
        using e1 e2 LET
        by (smt (verit) DiffD2 Diff_subset free_tv_W free_tv_gen_cons
            free_tv_le_new_tv new_tv_W subsetD weaken_A_Int_B_eq_empty)
```

```
      then
      show "gen ($ (λa. $ S2 (S1 a)) A) ($ S2 t1) # $ (λa. $ S2 (S1 a)) A ⊢ e2 :: t"
        using e1 e2 LET
        by (metis app_subst_Cons gen_subst_commutes new_tv_Cons new_tv_W new_tv_compatible_W
            new_tv_compatible_gen new_tv_subst_scheme_list subst_comp_scheme_list)
    qed
  qed
qed
```

— Completeness of W w.r.t. *has_type*

```
lemma W_complete_lemma:
  "⟦$S' A ⊢ e :: t'; new_tv n A⟧ ⟹
   ∃S t. (∃m. W e A n = Some (S,t,m)) ∧ (∃R. $S' A = $R ($S A) ∧ t' = $R t)"
proof (induction e arbitrary: S' A t' n)
  case (Var u) thus ?case
  proof (clarsimp simp add: has_type_simps is_bound_typ_instance)
    fix S :: "nat ⇒ typ"
    assume A: "new_tv n A" "u < length A"
    show "∃R. $ S' A = $ R A ∧
      bound_typ_inst S ($ S' A ! u) = $ R (bound_typ_inst (λb. TVar (b + n)) (A ! u))"
    proof (intro exI conjI)
      show "$ S' A = $ (λx. if x < n then S' x else S (x - n)) A"
        using Var.prems(2) new_if_subst_type_scheme_list by force
      show "bound_typ_inst S ($ S' A ! u) = $ (λx. if x < n then S' x else S (x - n))
(bound_typ_inst (λb. TVar (b + n)) (A ! u))"
        using A
        by (simp add: new_if_subst_type_scheme  new_tv_nth_nat_A o_def nth_subst
                flip: bound_typ_inst_composed_subst)
    qed
  qed
next
  case (Abs e S' A t' n)
  then obtain t1 t2 where "t' = t1 -> t2" "mk_scheme t1 # $ S' A ⊢ e :: t2"
    by (auto simp: has_type_simps)
  with Abs.prems Abs.IH[of "λx. if x=n then t1 else (S' x)" "(FVar n) #A" t2 "Suc n"]
  show ?case
    by (force dest!: mk_scheme_injective)
next
  case (App e1 e2)
  then obtain t2 where e2t: "$ S' A ⊢ e2 :: t2"  and e1t: "$ S' A ⊢ e1 :: t2 -> t'"
    by (auto simp: has_type_simps)
  then obtain S t m R
    where e1: "W e1 A n = Some (S, t, m)" and R: "$ S' A = $ R ($ S A)" "t2 -> t' = $
R t"
    using App by blast
  with App.prems have new_tv_m: "new_tv m ($ S A)"
    by (metis new_tv_W new_tv_compatible_W new_tv_subst_scheme_list)
  with App R
  obtain Sa ta ma Ra where We2: "W e2 ($ S A) m = Some (Sa, ta, ma)"
```

```
        and RSA: "$ R ($ S A) = $ Ra ($ Sa ($ S A))"
        and t2eq: "t2 = $ Ra ta"
  by (metis e2t)
define F where "F ≡ (λx. if x = ma then t'
                         else if x ∈ free_tv t - free_tv Sa then R x
                         else Ra x)"
have "ma ∉ free_tv t"
  by (metis App.prems(2) W_var_geD We2 e1 new_tv_W new_tv_le
      new_tv_not_free_tv)
have "$ F (Sa na) = R na" if "na ∈ free_tv t" for na
proof -
  have "na ≠ ma"
    using ‹ma ∉ free_tv t› that by auto
  show ?thesis
  proof (cases "na ∈ free_tv Sa")
    case True
    then have "R na = $ Ra (Sa na)"
      by (metis (lifting) App.prems(2) RSA We2 e1 eq_subst_scheme_list_eq_free free_tv_W
          free_tv_le_new_tv new_tv_W subst_comp_scheme_list that)
    then show ?thesis
      by (metis F_def True We2 new_tv_m codD cod_app_subst eq_free_eq_subst_te
          new_tv_W new_tv_not_free_tv weaken_not_elem_A_minus_B)
  next
    case False
    then show ?thesis
      using not_free_impl_id [OF False] ‹na ≠ ma› that
      by (simp add: F_def)
  qed
qed
then have *: "$ F ($ Sa t) = $ Ra ta -> t'"
  using eq_free_eq_subst_te subst_comp_te using R t2eq by fastforce
moreover have "Ra na = F na"
  if "na ∈ free_tv ta" for na
proof -
  have "na ≠ ma"
    using We2 new_tv_W new_tv_m new_tv_not_free_tv that by blast
  show ?thesis
  proof (cases "na ∈ free_tv t - free_tv Sa")
    case True
    then have "$ R ($ S A) = $ (λx. $ Ra (Sa x)) ($ S A)"
      by (metis RSA subst_comp_scheme_list)
    then have "Ra na = R na"
      by (metis that App.prems(2) DiffE True Type.app_subst_TVar We2 free_tv_W e1
          eq_subst_scheme_list_eq_free free_tv_le_new_tv new_tv_W not_free_impl_id)
    with ‹na ≠ ma› True show ?thesis
      by (simp add: F_def)
  next
    case False
    then show ?thesis
```

```
        using F_def ‹na ≠ ma› by presburger
      qed
    qed
    ultimately have "$ F ($ Sa t) = $ F (ta -> (TVar ma))"
      by (metis eq_free_eq_subst_te F_def Type.app_subst_Fun Type.app_subst_TVar)
    with mgu_Some obtain Sx Rb where Sx: "mgu ($ Sa t) (ta -> TVar ma) = Some Sx"
      and Rb: "F = $ Rb ∘ Sx"
      using mgu_mg by blast
    have t': "t' = $ Rb (Sx ma)"
      by (metis F_def Rb comp_def)
    have "$ Ra ($ Sa ($ S A)) = $ (λx. $ Rb (Sx x)) ($ Sa ($ S A))"
    proof (intro eq_free_eq_subst_scheme_list)
      fix na :: nat
      assume na: "na ∈ free_tv ($ Sa ($ S A))"
      then have "ma ≠ na"
        by (metis We2 new_tv_W new_tv_compatible_W new_tv_m new_tv_not_free_tv
            new_tv_subst_scheme_list)
      show "Ra na = $ Rb (Sx na)"
      proof (cases "na ∈ free_tv t - free_tv Sa")
        case True
        then have "na ∈ cod Sa ∪ free_tv ($ S A)"
          using free_tv_app_subst_scheme_list na by blast
        with ‹ma ≠ na› Rb show ?thesis
          by (smt (verit, ccfv_SIG) DiffD2 F_def RSA Rb Type.app_subst_TVar Un_iff codD
              comp_apply eq_subst_scheme_list_eq_free not_free_impl_id subst_comp_scheme_list)
      next
        case False
        then show ?thesis
          by (metis F_def Rb ‹ma ≠ na› comp_apply)
      qed
    qed
    then have "$ S' A = $ Rb ($ ($ Sx ∘ $ Sa ∘ S) A)"
      by (metis (no_types, lifting) ext R(1) RSA comp_apply fun.map_comp
          subst_comp_scheme_list)
    with We2 Sx show ?case
      by (auto simp add: e1 t')
  next
    case (LET e1 e2)
    then obtain t1 where t1: "$ S' A ⊢ e1 :: t1" "gen ($ S' A) t1 # $ S' A ⊢ e2 :: t'"
      by (auto simp: has_type_simps)
    then obtain S t m R where e1: "W e1 A n = Some (S, t, m)" "$ S' A = $ R ($ S A)"
        and "gen ($ R ($ S A)) ($ R t) # $ R ($ S A) ⊢ e2 :: t'"
      using LET by metis
    then have "$ R (gen ($ S A) t) # $ R ($ S A) ⊢ e2 :: t'"
      using gen_bound_typ_instance has_type_le_env le_env_Cons le_env_refl
      by presburger
    moreover
    have "new_tv m (gen ($ S A) t) ∧ new_tv m ($ S A)"
      using LET.prems e1
```

```
    by (metis new_tv_W new_tv_compatible_W new_tv_compatible_gen new_tv_subst_scheme_list)
  ultimately show ?case
    using LET.IH(2)[of R "gen ($ S A) t # $ S A" t' m] e1 subst_comp_scheme_list
    by auto
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.