Type inference for let-free MiniML

Dieter Nazareth, Tobias Nipkow, Thomas Stauner, Markus Wenzel

November 22, 2007

Contents

theory *W0*
imports *Main*
begin

1 Universal error monad

datatype $'a\ maybe = Ok\ 'a \mid Fail$

definition
$bind :: 'a\ maybe \Rightarrow ('a \Rightarrow 'b\ maybe) \Rightarrow 'b\ maybe$ (**infixl** $\mathsf{bind}\ 60$) **where**
$m\ \mathsf{bind}\ f = (case\ m\ of\ Ok\ r \Rightarrow f\ r \mid Fail \Rightarrow Fail)$

syntax
$\text{-}bind :: patterns \Rightarrow 'a\ maybe \Rightarrow 'b \Rightarrow 'c$ $((\text{-} := \text{-};//\text{-})\ 0)$
translations
$P := E;\ F == E\ \mathsf{bind}\ (\lambda P.\ F)$

lemma *bind-Ok* [*simp*]: $(Ok\ s)\ \mathsf{bind}\ f = (f\ s)$
by (*simp add*: *bind-def*)

lemma *bind-Fail* [*simp*]: *Fail* bind *f* = *Fail*
by (*simp add*: *bind-def*)

lemma *split-bind*:
$P\ (res\ \mathsf{bind}\ f) = ((res = Fail \longrightarrow P\ Fail) \wedge (\forall s.\ res = Ok\ s \longrightarrow P\ (f\ s)))$
by (*induct res*) *simp-all*

lemma *split-bind-asm*:
$P\ (res\ \mathsf{bind}\ f) = (\neg\ (res = Fail \wedge \neg\ P\ Fail \vee (\exists s.\ res = Ok\ s \wedge \neg\ P\ (f\ s))))$
by (*simp split*: *split-bind*)

lemmas *bind-splits* = *split-bind split-bind-asm*

lemma *bind-eq-Fail* [*simp*]:
$((m\ \mathsf{bind}\ f) = Fail) = ((m = Fail) \vee (\exists p.\ m = Ok\ p \wedge f\ p = Fail))$
by (*simp split*: *split-bind*)

lemma *rotate-Ok*: $(y = Ok\ x) = (Ok\ x = y)$
by (*rule eq-sym-conv*)

2 MiniML-types and type substitutions

axclass *type-struct* ⊆ *type*
— new class for structures containing type variables

datatype *typ* = *TVar nat* | *TFun typ typ* (**infixr** -> *70*)
— type expressions

types *subst* = *nat* => *typ*
— type variable substitution

instance *typ* :: *type-struct* **..**
instance *list* :: (*type-struct*) *type-struct* **..**
instance *fun* :: (*type*, *type-struct*) *type-struct* **..**

2.1 Substitutions

consts
app-subst :: *subst* ⇒ *'a*::*type-struct* ⇒ *'a*::*type-struct* ($)
— extension of substitution to type structures
primrec (*app-subst-typ*)
app-subst-TVar: $s (*TVar n*) = *s n*
app-subst-Fun: $s (*t1* -> *t2*) = $s *t1* -> $s *t2*

defs (**overloaded**)
app-subst-list: $s ≡ *map* ($s)

consts

$free\text{-}tv :: {}'a::type\text{-}struct \Rightarrow nat\ set$
— *free-tv* s: the type variables occuring freely in the type structure s

primrec (*free-tv-typ*)
$free\text{-}tv\ (TVar\ m) = \{m\}$
$free\text{-}tv\ (t1\ {-}{>}\ t2) = free\text{-}tv\ t1 \cup free\text{-}tv\ t2$

primrec (*free-tv-list*)
$free\text{-}tv\ [] = \{\}$
$free\text{-}tv\ (x\ \#\ xs) = free\text{-}tv\ x \cup free\text{-}tv\ xs$

definition
$dom :: subst \Rightarrow nat\ set$ **where**
$dom\ s = \{n.\ s\ n \neq TVar\ n\}$
— domain of a substitution

definition
$cod :: subst \Rightarrow nat\ set$ **where**
$cod\ s = (\bigcup m \in dom\ s.\ free\text{-}tv\ (s\ m))$
— codomain of a substitutions: the introduced variables

defs (**overloaded**)
free-tv-subst: $free\text{-}tv\ s \equiv dom\ s \cup cod\ s$

new-tv s n checks whether n is a new type variable wrt. a type structure s, i.e. whether n is greater than any type variable occuring in the type structure.

definition
$new\text{-}tv :: nat \Rightarrow {}'a::type\text{-}struct \Rightarrow bool$ **where**
$new\text{-}tv\ n\ ts = (\forall m.\ m \in free\text{-}tv\ ts \longrightarrow m < n)$

2.1.1 Identity substitution

definition
$id\text{-}subst :: subst$ **where**
$id\text{-}subst = (\lambda n.\ TVar\ n)$

lemma *app-subst-id-te* [*simp*]:
$\$id\text{-}subst = (\lambda t::typ.\ t)$
— application of *id-subst* does not change type expression
proof
fix $t :: typ$
show $\$id\text{-}subst\ t = t$
by (*induct t*) (*simp-all add*: *id-subst-def*)
qed

lemma *app-subst-id-tel* [*simp*]: $\$id\text{-}subst = (\lambda ts::typ\ list.\ ts)$
— application of *id-subst* does not change list of type expressions
proof

fix *ts* :: *typ list*
show $\$id\text{-}subst\ ts = ts$
by (*induct ts*) (*simp-all add*: *app-subst-list*)
qed

lemma *o-id-subst* [*simp*]: $\$s\ o\ id\text{-}subst = s$
by (*rule ext*) (*simp add*: *id-subst-def*)

lemma *dom-id-subst* [*simp*]: $dom\ id\text{-}subst = \{\}$
by (*simp add*: *dom-def id-subst-def*)

lemma *cod-id-subst* [*simp*]: $cod\ id\text{-}subst = \{\}$
by (*simp add*: *cod-def*)

lemma *free-tv-id-subst* [*simp*]: $free\text{-}tv\ id\text{-}subst = \{\}$
by (*simp add*: *free-tv-subst*)

lemma *cod-app-subst* [*simp*]:
assumes *free*: $v \in free\text{-}tv\ (s\ n)$
and *neq*: $v \neq n$
shows $v \in cod\ s$
proof −
have $s\ n \neq TVar\ n$
proof
assume $s\ n = TVar\ n$
with *free* **have** $v = n$ **by** *simp*
with *neq* **show** *False* **..**
qed
with *free* **show** *?thesis*
by (*auto simp add*: *dom-def cod-def*)
qed

lemma *subst-comp-te*: $\$g\ (\$f\ t :: typ) = \$(\lambda x.\ \$g\ (f\ x))\ t$
— composition of substitutions
by (*induct t*) *simp-all*

lemma *subst-comp-tel*: $\$g\ (\$f\ ts :: typ\ list) = \$(\lambda x.\ \$g\ (f\ x))\ ts$
by (*induct ts*) (*simp-all add*: *app-subst-list subst-comp-te*)

lemma *app-subst-Nil* [*simp*]: $\$s\ [] = []$
by (*simp add*: *app-subst-list*)

lemma *app-subst-Cons* [*simp*]: $\$s\ (t\ \#\ ts) = (\$s\ t)\ \#\ (\$s\ ts)$
by (*simp add*: *app-subst-list*)

lemma *new-tv-TVar* [*simp*]: $new\text{-}tv\ n\ (TVar\ m) = (m < n)$
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 add*: *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* (*t* # *ts*) = (*new-tv n t* ∧ *new-tv n ts*)
by (*auto simp add*: *new-tv-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-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 (*tactic safe-tac HOL-cs*)
— ⟹
apply (*tactic* ⟪ *fast-tac* (*HOL-cs addDs* [@{*thm leD*}] *addss* (*simpset*()
addsimps [*thm free-tv-subst*, *thm dom-def*])) *1* ⟫)
apply (*subgoal-tac m* ∈ *cod s* ∨ *s l* = *TVar l*)
apply (*tactic safe-tac HOL-cs*)
apply (*tactic* ⟪ *fast-tac* (*HOL-cs addDs* [*UnI2*] *addss* (*simpset*()
addsimps [*thm free-tv-subst*])) *1* ⟫)
apply (*drule-tac P* = *λx*. *m* ∈ *free-tv x* **in** *subst*, *assumption*)
apply *simp*
apply (*tactic* ⟪ *fast-tac* (*set-cs addss* (*simpset*()
addsimps [*thm free-tv-subst*, *thm cod-def*, *thm dom-def*])) *1* ⟫)
— ⟸
apply (*unfold free-tv-subst cod-def dom-def*)
apply *safe*
apply (*metis linorder-not-less*)+
done

lemma *new-tv-list*: *new-tv n x* = (∀ *y* ∈ *set x*. *new-tv n y*)
by (*induct x*) *simp-all*

lemma *subst-te-new-tv* [*simp*]:
new-tv n (*t*::*typ*) ⟹ $(*λx*. *if x* = *n then t′ else s x*) *t* = $*s t*
— substitution affects only variables occurring freely
by (*induct t*) *simp-all*

lemma *subst-tel-new-tv* [*simp*]:
new-tv n (*ts*::*typ list*) ⟹ $(*λx*. *if x* = *n then t else s x*) *ts* = $*s ts*
by (*induct ts*) *simp-all*

```
lemma new-tv-le: n ≤ m ⟹ new-tv n (t::typ) ⟹ new-tv m t
  — all greater variables are also new
proof (induct t)
  case (TVar n)
  then show ?case by (auto intro: less-le-trans)
next
  case TFun
  then show ?case by simp
qed

lemma [simp]: new-tv n t ⟹ new-tv (Suc n) (t::typ)
  by (rule lessI [THEN less-imp-le [THEN new-tv-le]])

lemma new-tv-list-le:
  assumes n ≤ m
  shows new-tv n (ts::typ list) ⟹ new-tv m ts
proof (induct ts)
  case Nil
  then show ?case by simp
next
  case Cons
  with ‹n ≤ m› show ?case by (auto intro: new-tv-le)
qed

lemma [simp]: new-tv n ts ⟹ new-tv (Suc n) (ts::typ list)
  by (rule lessI [THEN less-imp-le [THEN new-tv-list-le]])

lemma new-tv-subst-le: n ≤ m ⟹ new-tv n (s::subst) ⟹ new-tv m s
  apply (simp add: new-tv-subst)
  apply clarify
  apply (rule-tac P = l < n and Q = n <= l in disjE)
    apply clarify
    apply (simp-all add: new-tv-le)
  done

lemma [simp]: new-tv n s ⟹ new-tv (Suc n) (s::subst)
  by (rule lessI [THEN less-imp-le [THEN new-tv-subst-le]])

lemma new-tv-subst-var:
    n < m ⟹ new-tv m (s::subst) ⟹ new-tv m (s n)
  — new-tv property remains if a substitution is applied
  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-tel [simp]:
    new-tv n s ⟹ new-tv n (ts::typ list) ⟹ new-tv n ($s ts)
```

by (*induct ts*) (*fastsimp simp add*: *new-tv-subst*)+

lemma *new-tv-Suc-list*: *new-tv n ts --> new-tv* (*Suc n*) (*TVar n # ts*)
— auxilliary lemma
by (*simp add*: *new-tv-list*)

lemma *new-tv-subst-comp-1* [*simp*]:
$new\text{-}tv\ n\ (s::subst) \Longrightarrow new\text{-}tv\ n\ r \Longrightarrow new\text{-}tv\ n\ (\$r\ o\ s)$
— composition of substitutions preserves *new-tv* proposition
by (*simp add*: *new-tv-subst*)

lemma *new-tv-subst-comp-2* [*simp*]:
$new\text{-}tv\ n\ (s::subst) \Longrightarrow new\text{-}tv\ n\ r \Longrightarrow new\text{-}tv\ n\ (\lambda v.\ \$r\ (s\ v))$
by (*simp add*: *new-tv-subst*)

lemma *new-tv-not-free-tv* [*simp*]: $new\text{-}tv\ n\ ts \Longrightarrow n \notin free\text{-}tv\ ts$
— new type variables do not occur freely in a type structure
by (*auto simp add*: *new-tv-def*)

lemma *ftv-mem-sub-ftv-list* [*simp*]:
$(t::typ) \in set\ ts \Longrightarrow free\text{-}tv\ t \subseteq free\text{-}tv\ ts$
by (*induct ts*) *auto*

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 *eq-subst-te-eq-free*:
$\$s1\ (t::typ) = \$s2\ t \Longrightarrow n \in free\text{-}tv\ t \Longrightarrow s1\ n = s2\ n$
by (*induct t*) *auto*

lemma *eq-free-eq-subst-te*:
$(\forall n.\ n \in free\text{-}tv\ t\ {-}{-}{>}\ s1\ n = s2\ n) \Longrightarrow \$s1\ (t::typ) = \$s2\ t$
by (*induct t*) *auto*

lemma *eq-subst-tel-eq-free*:
$\$s1\ (ts::typ\ list) = \$s2\ ts \Longrightarrow n \in free\text{-}tv\ ts \Longrightarrow s1\ n = s2\ n$
by (*induct ts*) (*auto intro*: *eq-subst-te-eq-free*)

lemma *eq-free-eq-subst-tel*:
$(\forall n.\ n \in free\text{-}tv\ ts\ {-}{-}{>}\ s1\ n = s2\ n) \Longrightarrow \$s1\ (ts::typ\ list) = \$s2\ ts$
by (*induct ts*) (*auto intro*: *eq-free-eq-subst-te*)

Some useful lemmas.

lemma *codD*: $v \in cod\ s \Longrightarrow v \in free\text{-}tv\ s$
by (*simp add*: *free-tv-subst*)

lemma *not-free-impl-id*: $x \notin free\text{-}tv\ s \Longrightarrow s\ x = TVar\ x$
by (*simp add*: *free-tv-subst dom-def*)

lemma *free-tv-le-new-tv*: *new-tv n t* $\Longrightarrow$ *m* $\in$ *free-tv t* $\Longrightarrow$ *m* $<$ *n*
by (*unfold new-tv-def*) *fast*

lemma *free-tv-subst-var*: *free-tv* (*s* (*v*::*nat*)) $\leq$ *insert v* (*cod s*)
by (*cases v* $\in$ *dom s*) (*auto simp add*: *cod-def dom-def*)

lemma *free-tv-app-subst-te*: *free-tv* (\$*s* (*t*::*typ*)) $\subseteq$ *cod s* $\cup$ *free-tv t*
by (*induct t*) (*auto simp add*: *free-tv-subst-var*)

lemma *free-tv-app-subst-tel*: *free-tv* (\$*s* (*ts*::*typ list*)) $\subseteq$ *cod s* $\cup$ *free-tv ts*
apply (*induct ts*)
apply *simp*
apply (*cut-tac free-tv-app-subst-te*)
apply *fastsimp*
done

lemma *free-tv-comp-subst*:
free-tv (λ*u*::*nat*. \$*s1* (*s2 u*) :: *typ*) $\subseteq$ *free-tv s1* $\cup$ *free-tv s2*
apply (*unfold free-tv-subst dom-def*)
apply (*tactic* ⟪
fast-tac (*set-cs addSDs* [*thm free-tv-app-subst-te RS subsetD*,
thm free-tv-subst-var RS subsetD]
addss (*simpset*() *delsimps* (*thms bex-simps*)
addsimps [*thm cod-def*, *thm dom-def*])) *1* ⟫)
done

2.2 Most general unifiers

consts
mgu :: *typ* $\Rightarrow$ *typ* $\Rightarrow$ *subst maybe*
axioms
mgu-eq [*simp*]: *mgu t1 t2* = *Ok u* $\Longrightarrow$ \$*u t1* = \$*u t2*
mgu-mg [*simp*]: *mgu t1 t2* = *Ok u* $\Longrightarrow$ \$*s t1* = \$*s t2* $\Longrightarrow$ $\exists$ *r*. *s* = \$*r o u*
mgu-Ok: \$*s t1* = \$*s t2* $\Longrightarrow$ $\exists$ *u*. *mgu t1 t2* = *Ok u*
mgu-free [*simp*]: *mgu t1 t2* = *Ok u* $\Longrightarrow$ *free-tv u* $\subseteq$ *free-tv t1* $\cup$ *free-tv t2*

lemma *mgu-new*: *mgu t1 t2* = *Ok u* $\Longrightarrow$ *new-tv n t1* $\Longrightarrow$ *new-tv n t2* $\Longrightarrow$ *new-tv n u*
— *mgu* does not introduce new type variables
by (*unfold new-tv-def*) (*blast dest*: *mgu-free*)

3 Mini-ML with type inference rules

datatype
expr = *Var nat* | *Abs expr* | *App expr expr*

Type inference rules.

inductive

```
  has-type :: typ list ⇒ expr ⇒ typ ⇒ bool  (((-) |−/ (-) :: (-)) [60, 0, 60] 60)
  where
    Var: n < length a ⟹ a |− Var n :: a ! n
  | Abs: t1#a |− e :: t2 ⟹ a |− Abs e :: t1 −> t2
  | App: a |− e1 :: t2 −> t1 ⟹ a |− e2 :: t2
      ⟹ a |− App e1 e2 :: t1
```

Type assigment is closed wrt. substitution.

```
lemma has-type-subst-closed: a |− e :: t ==> $s a |− e :: $s t
proof (induct set: has-type)
  case (Var n a)
  then have n < length (map ($ s) a) by simp
  then have map ($ s) a |− Var n :: map ($ s) a ! n
    by (rule has-type.Var)
  also have map ($ s) a ! n = $ s (a ! n)
    by (rule nth-map) (rule Var)
  also have map ($ s) a = $ s a
    by (simp only: app-subst-list)
  finally show ?case .
next
  case (Abs t1 a e t2)
  then have $ s t1 # map ($ s) a |− e :: $ s t2
    by (simp add: app-subst-list)
  then have map ($ s) a |− Abs e :: $ s t1 −> $ s t2
    by (rule has-type.Abs)
  then show ?case
    by (simp add: app-subst-list)
next
  case App
  then show ?case by (simp add: has-type.App)
qed
```

4 Correctness and completeness of the type inference algorithm W

```
consts
  𝒲 :: expr ⇒ typ list ⇒ nat ⇒ (subst × typ × nat) maybe
primrec
  𝒲 (Var i) a n =
    (if i < length a then Ok (id-subst, a ! i, n) else Fail)
  𝒲 (Abs e) a n =
    ((s, t, m) := 𝒲 e (TVar n # a) (Suc n);
     Ok (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);
     Ok ($u o $s2 o s1, $u (TVar m2), Suc m2))
```

```
theorem W-correct: Ok (s, t, m) = 𝒲 e a n ==> $s a |- e :: t
proof (induct e arbitrary: a s t m n)
  case (Var i)
  from ‹Ok (s, t, m) = 𝒲 (Var i) a n›
  show $s a |- Var i :: t by (simp add: has-type.Var split: if-splits)
next
  case (Abs e)
  from ‹Ok (s, t, m) = 𝒲 (Abs e) a n›
  obtain t' where t = s n -> t'
      and Ok (s, t', m) = 𝒲 e (TVar n # a) (Suc n)
    by (auto split: bind-splits)
  with Abs.hyps show $s a |- Abs e :: t
    by (force intro: has-type.Abs)
next
  case (App e1 e2)
  from ‹Ok (s, t, m) = 𝒲 (App e1 e2) a n›
  obtain s1 t1 n1 s2 t2 n2 u where
          s: s = $u o $s2 o s1
      and t: t = u n2
      and mgu-ok: mgu ($s2 t1) (t2 -> TVar n2) = Ok u
      and W1-ok: Ok (s1, t1, n1) = 𝒲 e1 a n
      and W2-ok: Ok (s2, t2, n2) = 𝒲 e2 ($s1 a) n1
    by (auto split: bind-splits simp: that)
  show $s a |- App e1 e2 :: t
  proof (rule has-type.App)
    from s have s': $u ($s2 ($s1 a)) = $s a
      by (simp add: subst-comp-tel o-def)
    show $s a |- e1 :: $u t2 -> t
    proof -
      from W1-ok have $s1 a |- e1 :: t1 by (rule App.hyps(1))
      then have $u ($s2 ($s1 a)) |- e1 :: $u ($s2 t1)
        by (intro has-type-subst-closed)
      with s' t mgu-ok show ?thesis by simp
    qed
    show $s a |- e2 :: $u t2
    proof -
      from W2-ok have $s2 ($s1 a) |- e2 :: t2 by (rule App.hyps(2))
      then have $u ($s2 ($s1 a)) |- e2 :: $u t2
        by (rule has-type-subst-closed)
      with s' show ?thesis by simp
    qed
  qed
qed


inductive-cases has-type-casesE:
  s |- Var n :: t
  s |- Abs e :: t
```

s |− *App e1 e2* ::*t*

lemmas [*simp*] = *Suc-le-lessD*
and [*simp del*] = *less-imp-le ex-simps all-simps*

lemma *W-var-ge* [*simp*]: !!*a n s t m*. $\mathcal{W}$ *e a n* = *Ok* (*s*, *t*, *m*) $\Longrightarrow$ *n* $\leq$ *m*
— the resulting type variable is always greater or equal than the given one
apply (*atomize* (*full*))
apply (*induct e*)

case *Var n*

apply *clarsimp*

case *Abs e*

apply (*simp split add*: *split-bind*)
apply (*fast dest*: *Suc-leD*)

case *App e1 e2*

apply (*simp* (*no-asm*) *split add*: *split-bind*)
apply (*intro strip*)
apply (*rename-tac s t na sa ta nb sb*)
apply (*erule-tac x* = *a* **in** *allE*)
apply (*erule-tac x* = *n* **in** *allE*)
apply (*erule-tac x* = $*s a* **in** *allE*)
apply (*erule-tac x* = *s* **in** *allE*)
apply (*erule-tac x* = *t* **in** *allE*)
apply (*erule-tac x* = *na* **in** *allE*)
apply (*erule-tac x* = *na* **in** *allE*)
apply (*simp add*: *eq-sym-conv*)
done

lemma *W-var-geD*: *Ok* (*s*, *t*, *m*) = $\mathcal{W}$ *e a n* $\Longrightarrow$ *n* $\leq$ *m*
by (*simp add*: *eq-sym-conv*)

lemma *new-tv-W*: !!*n a s t m*.
new-tv n a $\Longrightarrow$ $\mathcal{W}$ *e a n* = *Ok* (*s*, *t*, *m*) $\Longrightarrow$ *new-tv m s* & *new-tv m t*
— resulting type variable is new
apply (*atomize* (*full*))
apply (*induct e*)

case *Var n*

apply *clarsimp*
apply (*force elim*: *list-ball-nth simp add*: *id-subst-def new-tv-list new-tv-subst*)

case *Abs e*

apply (*simp* (*no-asm*) *add*: *new-tv-subst new-tv-Suc-list split add*: *split-bind*)
apply (*intro strip*)
apply (*erule-tac x* = *Suc n* **in** *allE*)

apply (*erule-tac x = TVar n # a* **in** *allE*)
apply (*fastsimp simp add*: *new-tv-subst new-tv-Suc-list*)

case *App e1 e2*

apply (*simp* (*no-asm*) *split add*: *split-bind*)
apply (*intro strip*)
apply (*rename-tac s t na sa ta nb sb*)
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 = na* **in** *allE*)
apply (*erule-tac x = na* **in** *allE*)
apply (*simp add*: *eq-sym-conv*)
apply (*erule-tac x = $s a* **in** *allE*)
apply (*erule-tac x = sa* **in** *allE*)
apply (*erule-tac x = ta* **in** *allE*)
apply (*erule-tac x = nb* **in** *allE*)
apply (*simp add*: *o-def rotate-Ok*)
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-subst-le*])
apply (*rule-tac n = na* **in** *new-tv-subst-le*)
apply (*simp add*: *rotate-Ok*)
apply (*simp* (*no-asm-simp*))
apply (*fast dest*: *W-var-geD intro*: *new-tv-list-le new-tv-subst-tel*
lessI [*THEN less-imp-le*, *THEN new-tv-subst-le*])
apply (*erule sym* [*THEN mgu-new*])
apply (*best dest*: *W-var-geD intro*: *new-tv-subst-te new-tv-list-le new-tv-subst-tel*
lessI [*THEN less-imp-le*, *THEN new-tv-le*] *lessI* [*THEN less-imp-le*, *THEN*
new-tv-subst-le]
new-tv-le)
apply (*tactic* ⟪ *fast-tac* (*HOL-cs addDs* [*thm W-var-geD*]
addIs [*thm new-tv-list-le*, *thm new-tv-subst-tel*, *thm new-tv-le*]
addss (*simpset*())) *1* ⟫)
apply (*rule lessI* [*THEN new-tv-subst-var*])
apply (*erule sym* [*THEN mgu-new*])
apply (*bestsimp intro*!: *lessI* [*THEN less-imp-le*, *THEN new-tv-le*] *new-tv-subst-te*
dest!: *W-var-geD intro*: *new-tv-list-le new-tv-subst-tel*
lessI [*THEN less-imp-le*, *THEN new-tv-subst-le*] *new-tv-le*)
apply (*tactic* ⟪ *fast-tac* (*HOL-cs addDs* [*thm W-var-geD*]
addIs [*thm new-tv-list-le*, *thm new-tv-subst-tel*, *thm new-tv-le*]
addss (*simpset*())) *1* ⟫)
done

lemma *free-tv-W*: !!*n a s t m v*. $\mathcal{W}$ *e a n* = *Ok* (*s*, *t*, *m*) $\Longrightarrow$
($v \in$ *free-tv s* $\vee$ $v \in$ *free-tv t*) $\Longrightarrow v < n \Longrightarrow v \in$ *free-tv a*
apply (*atomize* (*full*))

apply (*induct e*)

case *Var n*

apply *clarsimp*
apply (*tactic* ⟪ *fast-tac* (*HOL-cs addIs* [*thm nth-mem*, *subsetD*, *thm ftv-mem-sub-ftv-list*]) *1* ⟫)

case *Abs e*

apply (*simp add*: *free-tv-subst split add*: *split-bind*)
apply (*intro strip*)
apply (*rename-tac s t n1 v*)
apply (*erule-tac x* = *Suc n* **in** *allE*)
apply (*erule-tac x* = *TVar 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 (*force elim*!: *allE intro*: *cod-app-subst*)

case *App e1 e2*

apply (*simp* (*no-asm*) *split add*: *split-bind*)
apply (*intro strip*)
apply (*rename-tac s t n1 s1 t1 n2 s3 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*)

second case

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 (*tactic safe-tac* (*empty-cs addSIs* [*conjI*, *impI*] *addSEs* [*conjE*]))
apply (*simp add*: *rotate-Ok o-def*)
apply (*drule W-var-geD*)
apply (*drule W-var-geD*)
apply (*frule less-le-trans*, *assumption*)
apply (*fastsimp dest*: *free-tv-comp-subst* [*THEN subsetD*] *sym* [*THEN mgu-free*] *codD*
free-tv-app-subst-te [*THEN subsetD*] *free-tv-app-subst-tel* [*THEN subsetD*] *subsetD elim*: *UnE*)
apply *simp*
apply (*drule sym* [*THEN W-var-geD*])
apply (*drule sym* [*THEN W-var-geD*])

apply (*frule less-le-trans*, *assumption*)
apply (*tactic* ⟪ *fast-tac* (*HOL-cs addDs* [*thm mgu-free*, *thm codD*,
thm free-tv-subst-var RS subsetD,
thm free-tv-app-subst-te RS subsetD,
thm free-tv-app-subst-tel RS subsetD, @{*thm less-le-trans*}, *subsetD*]
addSEs [*UnE*] *addss* (*simpset*() *setSolver unsafe-solver*)) *1* ⟫)
— builtin arithmetic in simpset messes things up
done

Completeness of $\mathcal{W}$ wrt. *has-type*.

lemma *W-complete-aux*: $!!s'\ a\ t'\ n.\ \$s'\ a\ |\!-\ e :: t' \Longrightarrow$ *new-tv* $n\ a \Longrightarrow$
$(\exists\, s\ t.\ (\exists\, m.\ \mathcal{W}\ e\ a\ n = Ok\ (s,\ t,\ m)) \wedge (\exists\, r.\ \$s'\ a = \$r\ (\$s\ a) \wedge t' = \$r\ t))$
apply (*atomize* (*full*))
apply (*induct e*)

case *Var n*

apply (*intro strip*)
apply (*simp* (*no-asm*) *cong add*: *conj-cong*)
apply (*erule has-type-casesE*)
apply (*simp add*: *eq-sym-conv app-subst-list*)
apply (*rule-tac* $x = s'$ **in** *exI*)
apply *simp*

case *Abs e*

apply (*intro strip*)
apply (*erule has-type-casesE*)
apply (*erule-tac* $x = \lambda x.$ *if* $x = n$ *then t1 else* $(s'\ x)$ **in** *allE*)
apply (*erule-tac* $x = TVar\ n\ \#\ a$ **in** *allE*)
apply (*erule-tac* $x = t2$ **in** *allE*)
apply (*erule-tac* $x = Suc\ n$ **in** *allE*)
apply (*fastsimp cong add*: *conj-cong split add*: *split-bind*)

case *App e1 e2*

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 (*tactic safe-tac HOL-cs*)
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 (*tactic safe-tac HOL-cs*)
apply (*tactic* ⟪ *fast-tac* (*HOL-cs addIs* [*sym RS thm W-var-geD*,
thm new-tv-W RS conjunct1, *thm new-tv-list-le*, *thm new-tv-subst-tel*]) *1* ⟫)

apply (*subgoal-tac*
$\$(\lambda x.\ if\ x = ma\ then\ t'\ else\ (if\ x \in free\text{-}tv\ t - free\text{-}tv\ sa\ then\ r\ x$
$else\ ra\ x))\ (\$\ sa\ t) =$
$\$(\lambda x.\ if\ x = ma\ then\ t'\ else\ (if\ x \in free\text{-}tv\ t - free\text{-}tv\ sa\ then\ r\ x$
$else\ ra\ x))\ (ta\ {-}{>}\ (TVar\ ma)))$
apply (*rule-tac* [*2*] $t = \$(\lambda x.\ if\ x = ma\ then\ t'$
$else\ (if\ x \in (free\text{-}tv\ t - free\text{-}tv\ sa)\ then\ r\ x\ else\ ra\ x))\ (\$sa\ t)$ **and**
$s = (\$\ ra\ ta)\ {-}{>}\ t'$ **in** *ssubst*)
prefer *2*
apply (*simp add*: *subst-comp-te*)
apply (*rule eq-free-eq-subst-te*)
apply (*intro strip*)
apply (*subgoal-tac* $na \neq ma$)
prefer *2*
apply (*fast dest*: *new-tv-W sym* [*THEN W-var-geD*] *new-tv-not-free-tv new-tv-le*)
apply (*case-tac* $na \in free\text{-}tv\ sa$)

$na \notin free\text{-}tv\ sa$

prefer *2*
apply (*frule not-free-impl-id*)
apply *simp*

$na \in free\text{-}tv\ sa$

apply (*drule-tac* $ts1 = \$s\ a$ **and** $r = \$\ r\ (\$\ s\ a)$ **in** *subst-comp-tel* [*THEN* [*2*] *trans*])
apply (*drule-tac eq-subst-tel-eq-free*)
apply (*fast intro*: *free-tv-W free-tv-le-new-tv dest*: *new-tv-W*)
apply *simp*
apply (*case-tac* $na \in dom\ sa$)
prefer *2*

$na \neq dom\ sa$

apply (*simp add*: *dom-def*)

$na \in dom\ sa$

apply (*rule eq-free-eq-subst-te*)
apply (*intro strip*)
apply (*subgoal-tac* $nb \neq ma$)
prefer *2*
apply (*frule new-tv-W*, *assumption*)
apply (*erule conjE*)
apply (*drule new-tv-subst-tel*)
apply (*fast intro*: *new-tv-list-le dest*: *sym* [*THEN W-var-geD*])
apply (*fastsimp dest*: *new-tv-W new-tv-not-free-tv simp add*: *cod-def free-tv-subst*)
apply (*fastsimp simp add*: *cod-def free-tv-subst*)
prefer *2*
apply (*simp* (*no-asm*))
apply (*rule eq-free-eq-subst-te*)
apply (*intro strip*)

```
   apply (subgoal-tac na ≠ ma)
    prefer 2
    apply (frule new-tv-W, assumption)
    apply (erule conjE)
    apply (drule sym [THEN W-var-geD])
    apply (fast dest: new-tv-list-le new-tv-subst-tel new-tv-W new-tv-not-free-tv)
   apply (case-tac na ∈ free-tv t − free-tv sa)
    prefer 2
```

case $na \notin$ *free-tv t* $-$ *free-tv sa*

```
    apply simp
    defer
```

case $na \in$ *free-tv t* $-$ *free-tv sa*

```
    apply simp
    apply (drule-tac ts1 = $s a and r = $ r ($ s a) in subst-comp-tel [THEN [2]
trans])
    apply (drule eq-subst-tel-eq-free)
     apply (fast intro: free-tv-W free-tv-le-new-tv dest: new-tv-W)
    apply (simp add: free-tv-subst dom-def)
   prefer 2 apply fast
  apply (simp (no-asm-simp) split add: split-bind)
  apply (tactic safe-tac HOL-cs)
   apply (drule mgu-Ok)
   apply fastsimp
  apply (drule mgu-mg, assumption)
  apply (erule exE)
  apply (rule-tac x = rb in exI)
  apply (rule conjI)
   prefer 2
   apply (drule-tac x = ma in fun-cong)
   apply (simp add: eq-sym-conv)
  apply (simp (no-asm) add: o-def subst-comp-tel [symmetric])
  apply (rule subst-comp-tel [symmetric, THEN [2] trans])
  apply (simp add: o-def eq-sym-conv)
  apply (rule eq-free-eq-subst-tel)
  apply (tactic safe-tac HOL-cs)
  apply (subgoal-tac ma ≠ na)
   prefer 2
   apply (frule new-tv-W, assumption)
   apply (erule conjE)
   apply (drule new-tv-subst-tel)
    apply (fast intro: new-tv-list-le dest: sym [THEN W-var-geD])
   apply (frule-tac n = m in new-tv-W, assumption)
   apply (erule conjE)
   apply (drule free-tv-app-subst-tel [THEN subsetD])
  apply (tactic ⟪ fast-tac (set-cs addDs [sym RS thm W-var-geD, thm new-tv-list-le,
    thm codD, thm new-tv-not-free-tv]) 1 ⟫)
  apply (case-tac na ∈ free-tv t − free-tv sa)
```

prefer *2*

case $na \notin$ *free-tv* t $-$ *free-tv* sa

apply *simp*
defer

case $na \in$ *free-tv* t $-$ *free-tv* sa

apply *simp*
apply (*drule free-tv-app-subst-tel* [*THEN subsetD*])
apply (*fastsimp dest*: *codD subst-comp-tel* [*THEN* [*2*] *trans*]
eq-subst-tel-eq-free simp add: *free-tv-subst dom-def*)
apply *fast*
done

lemma *W-complete*: [] |− e :: t' ==>
$\exists\, s\ t.\ (\exists\, m.\ \mathcal{W}\ e\ []\ n = Ok\ (s,\ t,\ m)) \wedge (\exists\, r.\ t' = \$r\ t)$
apply (*cut-tac* $a = []$ **and** $s' =$ *id-subst* **and** $e = e$ **and** $t' = t'$ **in** *W-complete-aux*)
apply *simp-all*
done

5 Equivalence of W and I

Recursive definition of type inference algorithm $\mathcal{I}$ for Mini-ML.

consts
$\mathcal{I}$ :: *expr* ⇒ *typ list* ⇒ *nat* ⇒ *subst* ⇒ (*subst* × *typ* × *nat*) *maybe*
primrec
$\mathcal{I}$ (*Var i*) *a n s* = (*if i* < *length a then Ok* (*s*, *a* ! *i*, *n*) *else Fail*)
$\mathcal{I}$ (*Abs e*) *a n s* = ((*s*, *t*, *m*) := $\mathcal{I}$ *e* (*TVar n* # *a*) (*Suc n*) *s*;
Ok (*s*, *TVar n* −> *t*, *m*))
$\mathcal{I}$ (*App e1 e2*) *a n s* =
((*s1*, *t1*, *m1*) := $\mathcal{I}$ *e1 a n s*;
(*s2*, *t2*, *m2*) := $\mathcal{I}$ *e2 a m1 s1*;
u := *mgu* (\$*s2 t1*) (\$*s2 t2* −> *TVar m2*);
Ok(\$*u o s2*, *TVar m2*, *Suc m2*))

Correctness.

lemma *I-correct-wrt-W*: !!*a m s s' t n*.
new-tv m a ∧ *new-tv m s* ⟹ $\mathcal{I}$ *e a m s* = *Ok* (*s'*, *t*, *n*) ⟹
∃ *r*. $\mathcal{W}$ *e* (\$*s a*) *m* = *Ok* (*r*, \$*s' t*, *n*) ∧ *s'* = (\$*r o s*)
apply (*atomize* (*full*))
apply (*induct e*)

case *Var n*

apply (*simp add*: *app-subst-list split*: *split-if*)

case *Abs e*

apply (*tactic* ⟪ *asm-full-simp-tac*

(*simpset*() *setloop* (*split-inside-tac* [*thm split-bind*])) *1* ⟫)
apply (*intro strip*)
apply (*rule conjI*)
apply (*intro strip*)
apply (*erule allE*)+
apply (*erule impE*)
prefer *2* **apply** (*fastsimp simp add*: *new-tv-subst*)
apply (*tactic* ⟪ *fast-tac* (*HOL-cs addIs* [*thm new-tv-Suc-list RS mp*,
thm new-tv-subst-le, @{*thm less-imp-le*}, @{*thm lessI*}]) *1* ⟫)
apply (*intro strip*)
apply (*erule allE*)+
apply (*erule impE*)
prefer *2* **apply** (*fastsimp simp add*: *new-tv-subst*)
apply (*tactic* ⟪ *fast-tac* (*HOL-cs addIs* [*thm new-tv-Suc-list RS mp*,
thm new-tv-subst-le, @{*thm less-imp-le*}, @{*thm lessI*}]) *1* ⟫)

case *App e1 e2*

apply (*tactic* ⟪ *simp-tac* (*simpset* () *setloop* (*split-inside-tac* [*thm split-bind*])) *1*
⟫)
apply (*intro strip*)
apply (*rename-tac s1′ t1 n1 s2′ t2 n2 sa*)
apply (*rule conjI*)
apply *fastsimp*
apply (*intro strip*)
apply (*rename-tac s1 t1′ n1′*)
apply (*erule-tac x* = *a* **in** *allE*)
apply (*erule-tac x* = *m* **in** *allE*)
apply (*erule-tac x* = *s* **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* = *a* **in** *allE*)
apply (*erule-tac x* = *n1* **in** *allE*)
apply (*erule-tac x* = *s1′* **in** *allE*)
apply (*erule-tac x* = *s2′* **in** *allE*)
apply (*erule-tac x* = *t2* **in** *allE*)
apply (*erule-tac x* = *n2* **in** *allE*)
apply (*rule conjI*)
apply (*intro strip*)
apply (*rule notI*)
apply *simp*
apply (*erule impE*)
apply (*frule new-tv-subst-tel*, *assumption*)
apply (*drule-tac a* = $*s a* **in** *new-tv-W*, *assumption*)
apply (*fastsimp dest*: *sym* [*THEN W-var-geD*] *new-tv-subst-le new-tv-list-le*)
apply (*fastsimp simp add*: *subst-comp-tel*)
apply (*intro strip*)
apply (*rename-tac s2 t2′ n2′*)
apply (*rule conjI*)

apply (*intro strip*)
apply (*rule notI*)
apply *simp*
apply (*erule impE*)
apply (*frule new-tv-subst-tel*, *assumption*)
apply (*drule-tac a* = $s *a* **in** *new-tv-W*, *assumption*)
apply (*fastsimp dest*: *sym* [*THEN W-var-geD*] *new-tv-subst-le new-tv-list-le*)
apply (*fastsimp simp add*: *subst-comp-tel subst-comp-te*)
apply (*intro strip*)
apply (*erule* (*1*) *notE impE*)
apply (*erule* (*1*) *notE impE*)
apply (*erule exE*)
apply (*erule conjE*)
apply (*erule impE*)
apply (*frule new-tv-subst-tel*, *assumption*)
apply (*drule-tac a* = $s *a* **in** *new-tv-W*, *assumption*)
apply (*fastsimp dest*: *sym* [*THEN W-var-geD*] *new-tv-subst-le new-tv-list-le*)
apply (*erule* (*1*) *notE impE*)
apply (*erule exE conjE*)+
apply (*simp* (*asm-lr*) *add*: *subst-comp-tel subst-comp-te o-def*, (*erule conjE*)+, *hypsubst*)+
apply (*subgoal-tac new-tv n2 s* ∧ *new-tv n2 r* ∧ *new-tv n2 ra*)
apply (*simp add*: *new-tv-subst*)
apply (*frule new-tv-subst-tel*, *assumption*)
apply (*drule-tac a* = $s *a* **in** *new-tv-W*, *assumption*)
apply (*tactic safe-tac HOL-cs*)
apply (*bestsimp dest*: *sym* [*THEN W-var-geD*] *new-tv-subst-le new-tv-list-le*)
apply (*fastsimp dest*: *sym* [*THEN W-var-geD*] *new-tv-subst-le new-tv-list-le*)
apply (*drule-tac e* = *e1* **in** *sym* [*THEN W-var-geD*])
apply (*drule new-tv-subst-tel*, *assumption*)
apply (*drule-tac ts* = $s *a* **in** *new-tv-list-le*, *assumption*)
apply (*drule new-tv-subst-tel*, *assumption*)
apply (*bestsimp dest*: *new-tv-W simp add*: *subst-comp-tel*)
done

lemma *I-complete-wrt-W*: !!*a m s*.
new-tv m a ∧ *new-tv m s* ⟹ $\mathcal{I}$ *e a m s* = *Fail* ⟹ $\mathcal{W}$ *e* ($s *a*) *m* = *Fail*
apply (*atomize* (*full*))
apply (*induct e*)
apply (*simp add*: *app-subst-list*)
apply (*simp* (*no-asm*))
apply (*intro strip*)
apply (*subgoal-tac TVar m* # $s *a* = $s (*TVar m* # *a*))
apply (*tactic* ⟪ *asm-simp-tac* (*HOL-ss addsimps* [*thm new-tv-Suc-list*, @{*thm lessI*} *RS* @{*thm less-imp-le*} *RS thm new-tv-subst-le*]) *1* ⟫)
apply (*erule conjE*)
apply (*drule new-tv-not-free-tv* [*THEN not-free-impl-id*])
apply (*simp* (*no-asm-simp*))

```
  apply (simp (no-asm-simp))
  apply (intro strip)
  apply (erule exE)+
  apply (erule conjE)+
  apply (drule I-correct-wrt-W [COMP swap-prems-rl])
   apply fast
  apply (erule exE)
  apply (erule conjE)
  apply hypsubst
  apply (simp (no-asm-simp))
  apply (erule disjE)
   apply (rule disjI1)
   apply (simp (no-asm-use) add: o-def subst-comp-tel)
   apply (erule allE, erule allE, erule allE, erule impE, erule-tac [2] impE,
     erule-tac [2] asm-rl, erule-tac [2] asm-rl)
   apply (rule conjI)
    apply (fast intro: W-var-ge [THEN new-tv-list-le])
   apply (rule new-tv-subst-comp-2)
    apply (fast intro: W-var-ge [THEN new-tv-subst-le])
   apply (fast intro!: new-tv-subst-tel intro: new-tv-W [THEN conjunct1])
  apply (rule disjI2)
  apply (erule exE)+
  apply (erule conjE)
  apply (drule I-correct-wrt-W [COMP swap-prems-rl])
   apply (rule conjI)
   apply (fast intro: W-var-ge [THEN new-tv-list-le])
   apply (rule new-tv-subst-comp-1)
   apply (fast intro: W-var-ge [THEN new-tv-subst-le])
   apply (fast intro!: new-tv-subst-tel intro: new-tv-W [THEN conjunct1])
  apply (erule exE)
  apply (erule conjE)
  apply hypsubst
  apply (simp add: o-def subst-comp-te [symmetric] subst-comp-tel [symmetric])
  done

end
```