Type Inference: In & Out of Context

by Robert J. Simmons
and William Lovas

This document shows a side-by-side comparison of two algorithms for type inference with let-polymorphism.

The code on the left-hand side ("the imperative algorithm") is a implementation of the fairly standard imperative approach to polymorphic type inference. Unification is imperative, and is based on a union-find structure.

The code on the right-hand side ("the functional algorithm") is an ML adaptation of Gundry, McBride, and McKinna's 2010 MSFP paper "Type Inference In Context."

Both pieces of code infer types for MinML:

```
structure MinML = struct

  datatype oper
    = OpPlus
    | OpTimes
    | OpMinus

  type var = string

  datatype exp
    = ExInt of int
    | ExOp of exp * oper * exp
    | ExBool of bool
    | ExIf of exp * exp * exp
    | ExAbs of var * exp
    | ExApp of exp * exp
    | ExPair of exp * exp
    | ExFst of exp
    | ExSnd of exp
    | ExLet of var * exp * exp
    | ExVar of var

  (* Designates which terms should be type-generalized *)
  (* Note: could be generalized to "valuable" *)
  fun value (ExInt _ | ExBool _ | ExAbs _ | ExVar _) = true
    | value (ExOp _ | ExIf _ | ExApp _ | ExLet _) = false
    | value (ExPair (e1, e2)) = value e1 andalso value e2
    | value (ExFst _ | ExSnd _) = false
  val value : exp -> bool = value

end
```

Haskell idioms are generally translated into ML idioms – for instance, there is no state monoid. However, one idiom that was particularly useful was the combination of "cons lists" (which associate like ML lists) and "snoc lists" (which associate the other way around).

```
infixr 5 >>
infix 5 <<

structure Cons = struct
  datatype 'a list = Nil | >> of 'a * 'a list
  fun exists f Nil = false
    | exists f (x >> xs) = if f x then true else exists f xs
  fun map f Nil = Nil
    | map f (x >> xs) = f x >> map f xs
end

structure Snoc = struct
  datatype 'a list = Lin | << of 'a list * 'a
  fun all f Lin = true
    | all f (xs << x) = if f x then all f xs else false
end

open Cons
open Snoc
```

The code for this example is online:
typesafety.net/rfl/typebase.sml (basics)
typesafety.net/rfl/typenoctx.sml (imperative)
typesafety.net/rfl/typeinctx.sml (functional)

(version 1: october 14, 2010)
(version 2: november 18, 2010 – added version information and links to code)

Types and Contexts

```
structure ImperativeAlgorithm = struct

  open MinML

  (*** types ***)
  type ty_bvar = int

  datatype typ
    = TyInt
    | TyBool
    | TyArrow of typ * typ
    | TyProd  of typ * typ
    | TyEvar  of ty_evar
    | TyBvar  of ty_bvar

  withtype ty_evar = typ option ref

  fun fresh () = TyEvar (ref NONE)
  val fresh : unit -> typ = fresh

  (*** type schemes ***)
  datatype scheme = Scheme of ty_bvar List.list * typ

  (*** contexts ***)
  type exp_entry = var * scheme
  type context = exp_entry Snoc.list

  (*** utilities for dealing with options ***)
  infixr ||
  val op || = Option.getOpt
  infixr >>=
  fun m >>= k = Option.mapPartial k m

  fun assoc a [] = NONE
    | assoc a ((b,t)::bts) = if a = b
                             then SOME t
                             else assoc a bts

  (*** exceptions ***)
  exception Unify of typ * typ
  exception Occur
  exception Unbound of var
```

Both algorithms need types to have unifable variables (ty_evar) and variables bound in type schemas (ty_bvar).

The imperative algorithm uses references to implement evars as a union-find structure; the functional algorithm just uses integers as names. Notice the difference between the "fresh" function for both approaches.

The functional algorithm avoids substituting types for type variables, even when creating a type scheme – this means that the datatype for schemes is a bit more complex in the functional approach.

The imperative algorithm needs contexts only to store and look up term variables. In the functional algorithm the context is also where we store all the information contained in the imperative algorithm's union-find structure.

The functional algorithm uses de Bruijn levels to implement bound variables.

This is not a critical distinction between the two approaches.

Maps a function over all the types in a scheme. Used to turn ty_evars into ty_bvars when generalizing, and vice-versa when specializing.

```
structure FunctionalAlgorithm = struct

  open MinML

  (*** types ***)
  datatype ty_bvar = Z | S of ty_bvar

  datatype typ
    = TyInt
    | TyBool
    | TyArrow of typ * typ
    | TyProd  of typ * typ
    | TyEvar  of ty_evar
    | TyBvar  of ty_bvar

  withtype ty_evar = int

  val fresh = let val x = ref 0 in fn () => (x := !x + 1; !x) end
  val fresh : unit -> ty_evar = fresh

  (*** type schemes ***)
  datatype scheme
    = Type of typ
    | All of scheme
    | LetS of typ * scheme

  fun map_scheme f s =
    case s of
      Type t => Type (f t)
    | All s' => All (map_scheme f s')
    | LetS (t, s') => LetS (f t, map_scheme f s')

  (*** contexts ***)
  type typ_entry = ty_evar * typ option
  type exp_entry = var * scheme

  datatype entry = T of typ_entry | E of exp_entry | Sep
  type context = entry Snoc.list
  type suffix = typ_entry Cons.list

  infix 4 <><
  fun ctx <>< Cons.Nil = ctx
    | ctx <>< (entry >> suf) = ctx << T entry <>< suf
  val (op <><) : context * suffix -> context = (op <><)

  (*** exceptions ***)
  exception Unify of context * typ * typ
  exception Occur
  exception Unbound of var
```

Type Unification

```
  (*** occurs check ***)
  fun occurs ev (TyInt | TyBool | TyBvar _) = false
    | occurs ev (TyArrow (t1, t2)) = occurs ev t1 orelse occurs ev t2
    | occurs ev (TyProd (t1, t2)) = occurs ev t1 orelse occurs ev t2
    | occurs ev (TyEvar ev') = ev = ev'
  val occurs : ty_evar -> typ -> bool = occurs

  (*** weak-head normalization and path compression ***)
  (* a typ t is weak-head-normal if either:
       - it is not of the form TyEvar _, or
       - it is of the form TyEvar ev, where !ev = NONE. *)

  (* binds an evar to a typ, raising Occur if such binding would by cyclic.
     invariant: to (ev <- t), !ev must either be NONE or SOME (TyEvar _)
                (to allow path compression). *)
  infix <-
  fun ev <- t =
    if occurs ev t
    then raise Occur
    else ev := SOME t
  val (op <-) : ty_evar * typ -> unit = (op <-)

  (* find what an evar points to, if anything, with path compression.
     invariant: if [follow ev = SOME t], then t is weak-head normal.
           (NB: if [follow ev = NONE], then TyEvar ev is weak-head normal.) *)
  fun follow ev = !ev >>= (fn t => (ev <- normalize t; !ev))

  (* weak-head normalize a typ.
     invariant: if [normalize t = t'], then t' is weak-head normal. *)
  and normalize (t as TyEvar ev) = follow ev || t
    | normalize t = t

  val follow : ty_evar -> typ option = follow
  val normalize : typ -> typ = normalize
```

One of the primary differences between the two algorithms is how they fix-up state and avoid circularity when unification learns that a variable needs to be bound to a term.

```
  (*** occurs check ***)
  fun occurs a (TyInt | TyBool | TyBvar _) = false
    | occurs a (TyArrow (t1, t2)) = occurs a t1 orelse occurs a t2
    | occurs a (TyProd (t1, t2)) = occurs a t1 orelse occurs a t2
    | occurs a (TyEvar b) = a = b
  val occurs : ty_evar -> typ -> bool = occurs

  fun occurs_suffix a =
      Cons.exists (fn (_,NONE) => false | (_, SOME t) => occurs a t)
  val occurs_suffix : ty_evar -> suffix -> bool = occurs_suffix

  (*** solving for a variable, collecting dependency suffix ***)
  fun solve ctx (a, suf, t) =
    case ctx of
      Lin => raise Fail "variables out of context"
    | (ctx << T (c, d)) =>
      let in
        case (a = c, occurs c t orelse occurs_suffix c suf, d) of
          (true, true, _)         => raise Occur
        | (true, false, NONE)     => ctx <>< suf <>< (c, SOME t) >> Nil
        | (true, false, SOME s)   => unify (ctx <>< suf) (t, s) << T (c, d)
        | (false, true, _)        => solve ctx (a, (c, d) >> suf, t)
        | (false, false, _)       => solve ctx (a, suf, t) << T (c, d)
      end
    | (ctx << d) => solve ctx (a, suf, t) << d
```

"ev <- t" in the imperative algorithm is roughly equivalent to "solve ctx (ev, suf, t)" in the functional algorithm.

We need the suffix in the functional algorithm to drag the part of the context upon which t depends further out in the context; a somewhat analogous dragging is performed by weak-head normalization in the imperative algorithm. It is when this reorganization is impossible that we raise Occurs.

```
  (*** unification ***)
  fun unifyWhn (TyArrow (t1, u1), TyArrow (t2, u2)) =
      (unify (t1, t2); unify (u1, u2))
    | unifyWhn (TyProd (t1, u1), TyProd (t2, u2)) =
      (unify (t1, t2); unify (u1, u2))
    | unifyWhn (TyInt, TyInt) = ()
    | unifyWhn (TyBool, TyBool) = ()

    (* Variable/variable *)
    | unifyWhn (TyEvar (ev1 as ref NONE), TyEvar (ev2 as ref NONE)) =
      let val ev = ref NONE in
        ev1 <- TyEvar ev;
        ev2 <- TyEvar ev
      end

    (* Variable/constructor *)
    | unifyWhn (TyEvar (ev as ref NONE), t2) = ev <- t2
    | unifyWhn (t1, TyEvar (ev as ref NONE)) = ev <- t1

    (* Failure *)
    | unifyWhn ((TyEvar _, _) | (_, TyEvar _)) = raise Fail "not wh-normal"
    | unifyWhn ((TyBvar _, _) | (_, TyBvar _)) = raise Fail "not closed"
    | unifyWhn (t1, t2) = raise Unify (t1, t2)

  and unify (t1, t2) = unifyWhn (normalize t1, normalize t2)

  val unifyWhn : typ * typ -> unit = unifyWhn
  val unify : typ * typ -> unit = unify
```

Unification is similar; the primary difference is the result of the weak-head normalization used in the imperative algorithm.

The functional algorithm deals with two cases where an evar is reached but has actually already been assigned a meaning; this possibility is also why "solve" must be mutually recursive with "unify" in the functional algorithm. This is precisely what weak-head normalization avoids in the imperative algorithm.

```
  (*** unification ***)
  and unify ctx (TyArrow (t1, t2), TyArrow (s1, s2)) =
      unify (unify ctx (t1, s1)) (t2, s2)
    | unify ctx (TyProd (t1, t2), TyProd (s1, s2)) =
      unify (unify ctx (t1, s1)) (t2, s2)
    | unify ctx (TyInt, TyInt) = ctx
    | unify ctx (TyBool, TyBool) = ctx

    (* Variable/variable *)
    | unify (ctx << T (c, d)) (TyEvar a, TyEvar b) =
      let in
        case (c = a, c = b, d) of
          (true, true, _)       => ctx << T (c, d)
        | (true, false, NONE)   => ctx << T (c, SOME (TyEvar b))
        | (false, true, NONE)   => ctx << T (c, SOME (TyEvar a))
        | (true, false, SOME t) => unify ctx (t, TyEvar b) << T (c, d)
        | (false, true, SOME t) => unify ctx (TyEvar a, t) << T (c, d)
        | (false, false, _)     => unify ctx (TyEvar a, TyEvar b) << T (c, d)
      end
    | unify (ctx << d) (TyEvar a, TyEvar b) =
      unify ctx (TyEvar a, TyEvar b) << d
    | unify Lin (TyEvar a, TyEvar b) = raise Fail "variables out of context"

    (* Variable/constructor *)
    | unify ctx (TyEvar a, s) = solve ctx (a, Nil, s)
    | unify ctx (t, TyEvar b) = solve ctx (b, Nil, t)

    (* Failure *)
    | unify ctx (t, s) = raise Unify (ctx, t, s)

  val unify : context -> typ * typ -> context = unify
  val solve : context -> ty_evar * suffix * typ -> context = solve
```

Specialization & Generalization

```
  (*** specalizing a type scheme ***)
  fun specalize (Scheme (tvs, t)) =
    let
      fun substWhn s (t as (TyInt | TyBool | TyEvar _)) = t
        | substWhn s (TyArrow (t, u)) = TyArrow (subst s t, subst s u)
        | substWhn s (TyProd (t, u)) = TyProd (subst s t, subst s u)
        | substWhn s (t as TyBvar a) = assoc a s || t
      and subst s t = substWhn s (normalize t)
    in
      subst (List.map (fn tv => (tv, fresh ())) tvs) t
    end
  val specalize : scheme -> typ = specalize
```

Specialization, in both cases, mostly just involves a lot of opening binders and substituting in ty_evars for the ty_bvars. The imperative algorithm does a simultaneous substitution; the functional algorithm does it one-at-a-time.

```
  (*** specalizing a type scheme ***)
  fun open_ty a t =
    case t of
      TyArrow (t1, t2) => TyArrow (open_ty a t1, open_ty a t2)
    | TyProd (t1, t2) => TyProd (open_ty a t1, open_ty a t2)
    | TyBvar Z => TyEvar a
    | TyBvar (S bvar) => TyBvar bvar
    | _ => t
  val open_ty : ty_evar -> typ -> typ = open_ty

  fun specalize (ctx, s) =
    case s of
      Type ty => (ctx, ty)
    | All s' =>
      let val b = fresh ()
      in specalize (ctx << T (b, NONE), map_scheme (open_ty b) s') end
    | LetS (t, s') =>
      let val b = fresh ()
      in specalize (ctx << T (b, SOME t), map_scheme (open_ty b) s') end
  val specalize : context * scheme -> context * typ = specalize
```

```
  (*** generalizing a type scheme ***)
  fun generalize (ctx, t) =
    let
      fun generalizable ev =
          Snoc.all (fn (_, Scheme (_, t)) => not (occurs ev t)) ctx
      val k = ref 0
      fun nextTyBvar () = !k before k := !k + 1
      fun collectWhn (TyInt | TyBool | TyBvar _) = []
        | collectWhn (TyArrow (t, u)) = collect t @ collect u
        | collectWhn (TyProd (t, u)) = collect t @ collect u
        | collectWhn (TyEvar ev) =
          if generalizable ev then
            let val k = nextTyBvar ()
            in ev <- TyBvar k; [k]
            end
          else
            []
      (* collect : typ -> tyvar list *)
      (* side effect: instantiates generalizable evars with new tyvars.
         invariant: if [collect t = tvs], then t is closed w.r.t. tvs. *)
      and collect t = collectWhn (normalize t)
    in
      Scheme (collect t, t)
    end
  val generalize : context * typ -> scheme = generalize
```

The imperative algorithm decides what variables to generalize by looking through the context to see if a ty_evar is free in any current expression variables; if not, the variable should be generalized.

The functional algorithm lays down a separator before starting inference and then uses "skim" to generalize the context up to the separator. (The "solve" function will drag a type variable past the separator if it should not be generalized.)

```
  (*** generalizing a type scheme ***)
  fun close_ty a t =
    case t of
      TyArrow (t1, t2) => TyArrow (close_ty a t1, close_ty a t2)
    | TyProd (t1, t2) => TyProd (close_ty a t1, close_ty a t2)
    | TyBvar bvar => TyBvar (S bvar)
    | TyEvar b => if a = b then TyBvar Z else TyEvar b
    | _ => t
  val close_ty : ty_evar -> typ -> typ = close_ty

  (* Given a suffix and a type, generalize to a scheme. No context needed. *)
  fun bind suf t =
    case suf of
      Nil => Type t
    | ((a, NONE) >> suf) => All (map_scheme (close_ty a) (bind suf t))
    | ((a, SOME t') >> suf) => LetS (t', map_scheme (close_ty a) (bind suf t))
  val bind : suffix -> typ -> scheme = bind

  (* Skim off just the right suffix to generalize. *)
  fun skim (ctx, suf) =
    case ctx of
      Lin => raise Fail "variables out of context"
    | (ctx << Sep) => (ctx, suf)
    | (ctx << T d) => skim (ctx, d >> suf)
    | (ctx << E _) => raise Fail "unexpected term variable"
  val skim : context * suffix -> context * suffix = skim

  val generalize : context * typ -> context * scheme =
   fn (ctx, t) =>
    let val (ctx, suf) = skim (ctx, Nil) in (ctx, bind suf t) end
```

Type Inference

```
  (*** expression variables ***)
  fun lookup ctx x =
    case ctx of
      Lin => raise Unbound x
    | (ctx << (y, s)) => if x = y then s else lookup ctx x
  val lookup : context -> var -> scheme = lookup

  (*** inference ***)
  fun infer (ctx, e) =
    case e of
      ExInt _ => TyInt
    | ExBool _ => TyBool
    | ExOp (e1, _, e2) =>
      let
        val t1 = infer (ctx, e1)
        val t2 = infer (ctx, e2)
      in
        unify (t1, TyInt); unify (t2, TyInt);
        TyInt
      end
    | ExIf (e1, e2, e3) =>
      let
        val t1 = infer (ctx, e1)
        val t2 = infer (ctx, e2)
        val t3 = infer (ctx, e3)
      in
        unify (t1, TyBool); unify (t2, t3);
        t2
      end
    | ExAbs (x, e) =>
      let
        val t1 = fresh ()
        val t2 = infer (ctx << (x, Scheme ([], t1)), e)
      in
        TyArrow (t1, t2)
      end
    | ExApp (e1, e2) =>
      let
        val t1 = infer (ctx, e1)
        val t2 = infer (ctx, e2)
        val t = fresh ()
      in
        unify (t1, TyArrow (t2, t));
        t
      end
    | ExVar x => specalize (lookup ctx x)
    | ExLet (x, e1, e2) =>
      let
        val t1 = infer (ctx, e1)
        val pt1 = if value e1
                  then generalize (ctx, t1)
                  else Scheme ([], t1)
        val t2 = infer (ctx << (x, pt1), e2)
      in
        t2
      end
    | ExPair (e1, e2) =>
      let
        val t1 = infer (ctx, e1)
        val t2 = infer (ctx, e2)
      in
        TyProd (t1, t2)
      end
    | ExFst e =>
      let
        val t = infer (ctx, e)
        val t1 = fresh ()
        val t2 = fresh ()
      in
        unify (t, TyProd (t1, t2)); t1
      end
    | ExSnd e =>
      let
        val t = infer (ctx, e)
        val t1 = fresh ()
        val t2 = fresh ()
      in
        unify (t, TyProd (t1, t2)); t2
      end
  val infer : context * exp -> typ = infer

end
```

The actual type inference steps end up being quite similar; stateful unification in the imperative algorithm lines up nicely with threading through "ctx," which is essentially just a state object, in the functional algorithm.

Note: the functional "ExLet" case here is a bit inelegant in that it calls the "value" function twice, the first time to decide whether to put down a separator (because we're gonna generalize!) and the second time to actually perform the generalization.

The pair projection cases get cut off when I print this out on 11x17 paper, but they're boring anyway...

```
  (*** expression variables ***)
  fun lookup ctx x =
    case ctx of
      Lin => raise Unbound x
    | (ctx << E (y, s)) => if x = y then s else lookup ctx x
    | (ctx << _) => lookup ctx x
  val lookup : context -> var -> scheme = lookup

  fun extract ctx x =
    case ctx of
      Lin => raise Fail "missing name"
    | (ctx << E (y, s)) =>
      if x = y then ctx else raise Fail "out-of-order variable"
    | (ctx << T d) => extract ctx x << T d
    | (ctx << _) => raise Fail "bad context entry"
  val extract : context -> var -> context = extract

  (*** inference ***)
  fun infer (ctx, e) =
    case e of
      ExInt _ => (ctx, TyInt)
    | ExBool _ => (ctx, TyBool)
    | ExOp (e1, _, e2) =>
      let
        val (ctx, t1) = infer (ctx, e1)
        val (ctx, t2) = infer (ctx, e2)
      in
        (unify (unify ctx (t1, TyInt)) (t2, TyInt),
         TyInt)
      end
    | ExIf (e1, e2, e3) =>
      let
        val (ctx, t1) = infer (ctx, e1)
        val (ctx, t2) = infer (ctx, e2)
        val (ctx, t3) = infer (ctx, e3)
      in
        (unify (unify ctx (t1, TyBool)) (t2, t3),
         t3)
      end
    | ExAbs (x, e) =>
      let
        val a = fresh ()
        val (ctx, t') = infer (ctx << T (a, NONE) << E (x, Type (TyEvar a)), e)
      in
        (extract ctx x, TyArrow (TyEvar a, t'))
      end
    | ExApp (e1, e2) =>
      let
        val (ctx, t1) = infer (ctx, e1)
        val (ctx, t2) = infer (ctx, e2)
        val b = fresh ()
      in
        (unify (ctx << T (b, NONE)) (t1, TyArrow (t2, TyEvar b)),
         TyEvar b)
      end
    | ExVar x => specalize (ctx, lookup ctx x)
    | ExLet (x, e1, e2) =>
      let
        val (ctx, t1) = infer (if value e1 then ctx << Sep else ctx, e1)
        val (ctx, s1) = if value e1
                        then generalize (ctx, t1)
                        else (ctx, Type t1)
        val (ctx, t2) = infer (ctx << E (x, s1), e2)
      in
        (extract ctx x, t2)
      end
    | ExPair (e1, e2) =>
      let
        val (ctx, t1) = infer (ctx, e1)
        val (ctx, t2) = infer (ctx, e2)
      in
        (ctx, TyProd (t1, t2))
      end
    | ExFst e =>
      let
        val (ctx, t) = infer (ctx, e)
        val a = fresh ()
        val b = fresh ()
      in
        (unify ctx (t, TyProd (TyEvar a, TyEvar b)), TyEvar a)
      end
    | ExSnd e =>
      let
        val (ctx, t) = infer (ctx, e)
        val a = fresh ()
        val b = fresh ()
      in
        (unify ctx (t, TyProd (TyEvar a, TyEvar b)), TyEvar b)
      end
  val infer : context * exp -> context * typ = infer

end
```