LNCS 166, Proc Intl conf on Programming, Toulouse 1984.

Polymorphic Type Schemes and Recursive Definitions

Alan Mycroft
Programming Methodology Group
Institutionen för Informationsbehandling
Chalmers Tekniska Högskola
S-412 96 Göteborg, Sweden

Abstract: An extension to Milner's polymorphic type system is proposed and proved correct. Such an extension appears to be necessary for the class of languages with mutually recursive top-level definitions. We can now ascribe a more general type to such definitions than before.

1. Introduction

The polymorphic type system introduced in ML /GMW/ and formally proved correct by Milner /Mi/ has become popular. That this is so seems to be due to two factors. Firstly the polymorphism provides a type system which is sound (i.e. can detect all type errors) but without the irritating need to duplicate similar code at different types as occurs in Algol68 or Pascal: a function can be defined to operate on lists of type α rather than having to define separate functions for operating on lists of integers and on lists of booleans. (Incidentally Holmström /Ho/ demonstrates that a polymorphic program can be translated into a monomorphic one which uses a Pascal-like type system.) Secondly, the polymorphic type system can be used without user specified types and types are then inferred. This makes it useful for interactive work.

This popularity has brought the use of such type schemes into other languages, notably HOPE /BMS/ and Prolog /MO/. The problem is that the exemplified languages have a mutually recursive top level of definitions which, as implemented, non-trivially extend the ML type system without semantic justification. The problem we encounter is that in Milner's scheme the mutually recursive definition of map and squarelist in

$$\mathrm{map}(f,\ell) \;=\; \textit{if}\ \mathrm{null}(\ell)\ \textit{then}\ \ell\ \textit{else}\ \mathrm{cons}(f(\mathrm{hd}\ \ell), \mathrm{map}(f,\mathrm{tl}\ \ell))$$
$$\mathrm{squarelist}(\ell) \;=\; \mathrm{map}(\lambda x.x^2, \ell)$$

gives the types

$$\mathrm{map}:\ (\mathrm{int}\to\mathrm{int}) \times \mathrm{int\ list} \to \mathrm{int\ list}$$
$$\mathrm{squarelist}:\ \mathrm{int\ list} \to \mathrm{int\ list}$$

whereas their sequentially recursive definition (first of map, then of squarelist) gives the 'expected' type of

$$\mathrm{map}:\ \forall\alpha\beta\cdot(\alpha\to\beta) \times \alpha\ \mathrm{list} \to \beta\ \mathrm{list}$$

Worse still, if a third mutually recursive definition were to use map at a different type (e.g. bool list) then the three definitions could not be well-typed. This fact is seemingly not well-known and much reduces the usefulness of the type system for languages with such a feature. Although in the above example the type checker *could* determine that map and squarelist are not mutually recursive and so treat them as sequentially recursive definitions, we avoid such an idea since small changes in the program can drastically change the potential calling graph. Moreover this scheme fails to solve the underlying problem which also exists in ML; there are non-contrived examples associated with "object oriented" programming which fall foul of the restriction in a less avoidable manner and whose resolution requires duplication of functionally identical code. (See section 8.)

In the language Exp, introduced in section 2, we have a recursion operator in which the above definition can be written

let (map,squarelist) = *fix* (map,squarelist). $(\lambda(f,\ell). \ldots, \lambda\ell. \ldots)$.

We centre in on this and note that /Mi,DM/ give the same type rules for *fix* x.e as they would for FIX(λx.e) where FIX is assumed to be a predeclared function of type $\forall\alpha.(\alpha\to\alpha)\to\alpha$. Our solution is to give new, and more general, type rules for the former than the latter although, of course, they are intended to have the same semantics. In particular, we will allow different occurences of x in e to take on different instance types of that of x, subject to the types of x and e matching in a sense made precise later.

This idea is entirely parallel to the more general treatment of *let* x=e *in* e' compared with (λx.e')e which are semantically equivalent, but the first has more general type rules which allow x to take on different instance types in e' unlike the second. See /Mi/ for more discussion on this point which is closely related to the idea of generic and non-generic type variables.

Related work includes /MO/ in which the restriction on recursive definitions was first lifted for the special case of Prolog and /DMS/ in which certain definitions such as *fix* f. λx.f which we will consider ill-typed can be given the recursive (infinite) type $\mu\tau.\ \forall\alpha.\alpha\to\tau\ =\ \forall\alpha_0.\!\cdots\!.\alpha_0\to\alpha_1\to\cdots$. Note that they seek to give semantics to recursively defined types, whereas our aim is to give (finite) types to recursive definitions. This paper attempts to follow the notation of /DM/ who set the initial work of /Mi/ in a clearer framework and who sketched completeness. A completeness proof has also been given by /Ho/.

Sections 2 and 3 give the syntax of and operators on expressions and types. Section 4 follows by giving semantics for both and section 5 gives a semantically sound type inference system and proves the resulting inferrable types are principal. Section 6 uses unification to give a (semi-) algorithm for most general type assignment which is sound and complete for inference. This is followed by section 7 which gives an effective, though over-restrictive, condition to ensure termination.

2. The language

We follow /Mi/ and define the language Exp of expressions e to be given by the (abstract) syntax

e ::= x | e e' | λx.e | *fix* x.e | *let* x=e *in* e'

where x ranges over a set Id of identifiers. We omit Milner's *if* e *then* e' *else* e" construct since its effect (for type-checking purposes) is exactly that of the application IF e e' e" where IF is an identifier of type $\forall\alpha.bool\to\alpha\to\alpha\to\alpha$.

3. Types

Types are absent from the language Exp and we now introduce their syntax and operators. Discussion of their semantics occurs in section 4.

We assume a set TVar of *type variables* ranged over by α,β,γ and a set TCons of *type constructors* each with their arity. For simplicity we here assume that TCons = {*int*, *bool*,$\to$} having arity 0 except for $\to$ which has arity 2 and written infixed.

The set Type of *(simple) types*, ranged over by τ is given by the set of arity-respecting terms in the grammar

Type ::= TVar | TCons(Type,...,Type).

The set TScheme of *type schemes*, ranged over by σ is similarly given by

TScheme ::= Type | $\forall$TVar.TScheme.

It will be later useful to adjoin an element *err* to TScheme. *Monotypes* are types which do not contain type variables and are ranged over by μ. We have natural concepts of *free* and *bound* type variables. A type scheme is *closed* if it has no free type variables. Following /Mi,DM/ but not /MPS/ our type schemes have quantification ($\forall$) at the outermost level only.

A *(type) substitution* S is a finite map TVar$\to$Type often written $\{\tau_1/\alpha_1,\ldots,\tau_n/\alpha_n\}$. It is naturally extended to a map Type$\to$Type and, by acting on free variables only, to a map TScheme$\to$Tscheme. We say σ' is an *instance* of σ if $\sigma'=S\sigma$ for some substitution S.

We say $\sigma'=\forall\beta_1\ldots\beta_m.\tau'$ is a *generic instance* of $\sigma=\forall\alpha_1\cdots\alpha_n.\tau$ if there is a substitution S acting only on $\{\alpha_1\cdots\alpha_n\}$ such that $\tau'=S\tau$ and no β_i is free in σ. We write this as $\sigma\sqsubseteq\sigma'$ (/Mi/ uses $\sigma\geq\sigma'$). We naturally write $\sigma=\sigma'$ if $\sigma\sqsubseteq\sigma'\sqsubseteq\sigma$. Under this equivalence TScheme is a partial order with least element $\forall\alpha.\alpha$. It can be completed by adding the element *err* with $x\sqsubseteq err$. We will later consider monotonic functions on TScheme and it is convenient to draw part of it (fig 1). We note that in the $\sqsubseteq$ order type variables act like niladic type constructors and that infinite properly ascending chains have limit *err*. Moreover any subset X of TScheme has a *greatest lower bound* $\sqcap X$ with $\sqcap\{\} = err$. If X is a subset of TVar and $\sigma\in$TScheme we define $\overline{X}(\sigma) = \forall\alpha_1\cdots\alpha_n.\sigma$ where the α_i are free in σ but not in X. $\overline{X}$ is retractive on TScheme.

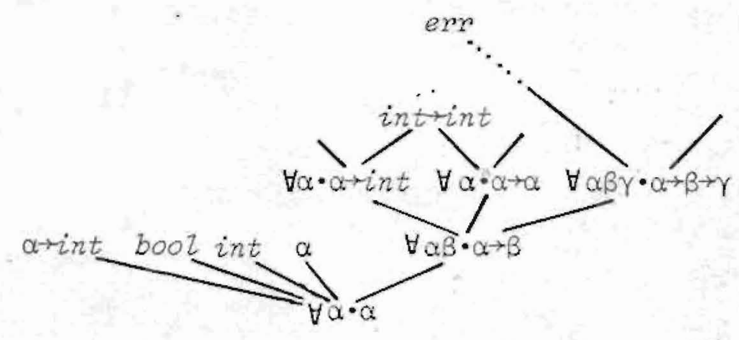

Figure 1: the cpo (TScheme, $\sqsubseteq$)

4. Semantics

This section defines the semantics of Exp and types. The interpreting domain for Exp will be given by V which satisfies the isomorphism

$$V = \mathbb{B} + \mathbb{Z} + (V \to V) + \{\mathit{wrong}\}_\bot$$

where $\mathbb{B}$ is the 3-element cpo of truth values, $\mathbb{Z}$ the cpo of integers with $\bot$ and + the coalesced sum. The three injection functions are called in_B, in_Z and in_F respectively.

We can now define the notion of *environment* Env, ranged over by η, as a finite (partial) map Id→V. Given such a η we define dom(η) to be the subset of Id on which it is defined. It is then standard that we define a semantic function

$$E\colon \ \text{Exp} \to \text{Env} \to V$$

in the obvious manner (see /Mi/).

We now follow /MPS/ and give closed type schemes a semantics in a similar manner. The meanings of types will naturally be *(left) ideals*, that is *downward closed and directed complete* /Mi,MPS/ subsets of V which do not contain *wrong*. The set of all such ideals is called I_V. The semantics of a closed type scheme σ is $T(\sigma)$ where $T\colon \text{TScheme} \to I_V$ is given by

$$\begin{aligned}
T[\![\, \mathit{bool} \,]\!] &= in_B(\,\mathbb{B}\,)\\
T[\![\, \mathit{int} \,]\!] &= in_Z(\,\mathbb{Z}\,)\\
T[\![\, \tau\to\tau' \,]\!] &= in_F\{f \in V\to V\colon\ f(T[\![\tau]\!]) \subseteq T[\![\tau']\!]\}\\
T[\![\, \sigma \,]\!] &\doteq \bigcap\{T[\![\mu]\!]\colon\ \sigma\le\mu,\ \mu \text{ monotype}\}
\end{aligned}$$

Lemma /MS/:

$$\sigma \le \sigma' \ \Rightarrow\ T[\![\sigma]\!] \subseteq T[\![\sigma']\!]$$

Following normal practice we define the space of *type assumptions* TA, ranged over by A, to be the set of finite maps Id → TScheme. A is *closed* if Ax is closed for all x in dom(A). We will write $A\{x{:}\sigma\}$ on type assumptions to stand for the usual $A\{\sigma/x\}$ which denotes the function agreeing with A except at x where its value is σ. By (helpful) abuse of notation we will define T on TA → $\mathcal{P}$(Env) by

$$T[\![A]\!] = \{\eta \in \text{Env}\colon\ \text{dom}(\eta)=\text{dom}(A),\ \forall x \in \text{dom}(\eta).\ \eta(x) \in T[\![Ax]\!]\}.$$

The atomic proposition $A \models e : \sigma$ is now defined. Intuitively it means that whenever e is evaluated with its free variables having values in types indicated by A then its result will have type σ. Formally it is defined by

$$A \models e{:}\sigma \iff \forall \eta \in T[\![A]\!].\ E[\![e]\!]\,\eta \ \in\ T[\![\sigma]\!]$$

provided A and σ are closed. Otherwise we define $A \models e{:}\sigma$ to be true iff all its closed instances are.

5. Type Inference

In this section we define a relation $\_ \vdash \_ : \_ \ \subseteq$ (TA x Exp x TScheme) which will enable us to deduce some true things about $\_ \models \_ : \_$. It is defined to be the least relation satisfying the following axioms. In this we follow /DM/, but the *fix* rule is new and discussed afterwards.

TAUT: $A \vdash x:\sigma \quad (\text{if } Ax=\sigma)$

SPEC: $\dfrac{A \vdash e:\sigma}{A \vdash e:\sigma'} \quad (\text{if } \sigma \sqsubseteq \sigma')$ GEN: $\dfrac{A \vdash e:\sigma}{A \vdash e: \forall\alpha.\sigma} \quad (\text{if } \alpha \text{ not free in } A)$

COMP: $\dfrac{A \vdash e: \tau'\to\tau \quad A \vdash e':\tau'}{A \vdash e\ e': \tau}$ ABS: $\dfrac{A\{x:\tau'\} \vdash e:\tau}{A \vdash \lambda x.e: \tau'\to\tau}$

FIX: $\dfrac{A\{x:\sigma\} \vdash e:\sigma}{A \vdash \mathit{fix}\ x.e: \sigma}$ LET: $\dfrac{A \vdash e:\sigma \quad A\{x:\sigma\} \vdash e':\tau}{A \vdash \mathit{let}\ x=e\ \mathit{in}\ e': \tau}$

In /Mi/ the FIX rule is given as (modulo change of notation)

FIX": $\dfrac{A\{x:\tau\} \vdash e:\tau}{A \vdash \mathit{fix}\ x.e: \tau}$

and /DM/ implicitly give the same rule by treating *fix* x.e as FIX(λx.e) where FIX is an identifier of type $\forall\alpha\cdot(\alpha\to\alpha)\to\alpha$. The proper generalisation (of FIX over FIX") is the basis of this work and enables the examples of the introduction to be typed in a natural way, since the type σ given to x in *fix* x.e can now be instantiated (with SPEC) at different occurrences of x within e. This extension is justified since it still results in only true things about $\models$ being $\vdash$ inferrable. Formally this is:

<u>Theorem</u> (semantic soundness)

For all A,e,σ we have $A \vdash e:\sigma \Rightarrow A \models e:\sigma$

<u>Proof</u>

/DM/ claim a proof by induction on e, to which we add the case for *fix* x.e. Assume, therefore, $A\{x:\sigma\} \vdash e:\sigma$, its implicant $A\{x:\sigma\} \models e:\sigma$. By the FIX rule we can deduce $A \vdash \mathit{fix}x.e:\sigma$ and hence we must show $A \models \mathit{fix}x.e:\sigma$.

Let $A' = A\{x:\sigma\}$ and η be an arbitrary member of $T[\![A]\!]$.

We have $E[\![\mathit{fix}\ x.e]\!]\eta = Y(\lambda v.E[\![e]\!]\eta\{v/x\}) = \bigsqcup_i v_i$

where $v_0 = \bot$ and $v_{i+1} = E[\![e]\!]\eta\{v_i/x\}$

By assumption $A' \models e:\sigma$ that is $\forall\eta' \in T[\![A']\!] . E[\![e]\!]\eta' \in T[\![\sigma]\!]$,

but we also have $v \in T[\![\sigma]\!] \Rightarrow \eta\{v/x\} \in T[\![A']\!]$ by definition of T

hence $v \in T[\![\sigma]\!] \Rightarrow E[\![e]\!]\eta\{v/x\} \in T[\![\sigma]\!]$.

So $v_0 = \bot \in T[\![\sigma]\!]$ and by the above $v_i \in T[\![\sigma]\!] \Rightarrow v_{i+1} \in T[\![\sigma]\!]$.

Hence $E[\![\mathit{fix}\ x.e]\!]\eta = \bigsqcup v_i \in T[\![\sigma]\!]$ by directed completeness of ideals.

Since η was arbitrary the last line holds for all η, which is just the definition of $A \models \mathit{fix}\ x.e:\sigma$ as required.

<u>Note:</u> To emphasise the point, if we are to have a computable set of types there can be no corresponding *semantic completeness*. When we come to discuss completeness it will be the *syntactic completeness* of an algorithm to infer instances of $A \vdash e:\sigma$.

As mentioned in section 3, we adjoin *err* to TScheme so it becomes a cpo with $\sqcap\{\} = err$. We still require $\sigma \in$ TScheme-$\{err\}$ for $A \vdash e{:}\sigma$ to hold.[†]

/DM/ show that the type inference rules (excepting our new FIX rule) are principal, i.e. for a given A and e, letting $\sigma = \sqcap\{\sigma' : A \vdash e{:}\sigma'\}$, we have

$\sigma \neq err \;\Rightarrow\; A \vdash e{:}\sigma$. ($\sigma$ is a principal type scheme for e in A.)

Of course, by the INST rule we also have

$\{\sigma' : A \vdash e{:}\sigma'\} \;=\; \{\sigma' : \sigma \sqsubseteq \sigma'\}$, which is a principal (right) ideal of TScheme-$\{err\}$

We now show that this result extends to the FIX rule, and derive a monotonic operator on TScheme used later. We prove the result by induction, assuming the e below contains at most $n \geq 0$ nested *fix* expressions and show it holds for n+1.

For a given $A \in TA$ and *fix* x.e $\in$ Exp, define the function

$$F_A^{x.e}: \text{TScheme} \to \text{TScheme}: \begin{cases} \sigma \;\to\; \sqcap\{\sigma' : A\{x{:}\sigma\} \vdash e{:}\sigma'\} \\ err \;\to\; err \end{cases}$$

We will often omit the sub- and super-script of F if the context is clear.

Lemma:

(i) F is monotonic and (ii) $F(\sigma) \neq err \Rightarrow A\{x{:}\sigma\} \vdash e{:}F(\sigma)$.

Proof:

(i) By lemma 1 of /DM/ we have that $\sigma_1 \sqsubseteq \sigma_2$ & $A\{x{:}\sigma_2\} \vdash e{:}\sigma' \Rightarrow A\{x{:}\sigma_1\} \vdash e{:}\sigma'$ by transforming derivations. The result follows from $X_1 \supseteq X_2 \Rightarrow \sqcap X_1 \sqsubseteq \sqcap X_2$.

(ii) By the principality of types for $A\{x{:}\sigma\}$ and e (inductive hypothesis).

Now, by the FIX inference rule, possibly followed by an INST rule we have:

Proposition 5.1:

$$A \vdash \textit{fix}\ x.e{:}\sigma \iff \sigma \sqsubseteq F_A^{x.e}(\sigma) \;\&\; \sigma \neq err$$

In other words the derivable types of *fix* x.e are just the non-*err* pre-fixpoints of F. Moreover the least fixpoint is the most general ($\sqsubseteq$ smallest) such σ and is expressible as $\bigsqcup F^i(\forall\alpha.\alpha)$ if this is non-*err*. If the limit is *err* then *fix*x.e has no deducible type under A. The former case gives a principal type scheme to *fix* x.e thus completing the inductive step. (In the latter case there is nothing to prove.)

Remark:

The induction over e could have been carried out without reference to the result of /DM/ and this would give us a characterisation of principality without reference to an algorithm for calculating principal types. (Principality is like confluence.)

The following proposition illustrates how the fixpoint iteration on types progresses and also shows that our approach treats the type of a recursive definition as the limit of types gained by expanding out the definition a finite number of times.

Proposition 5.2:

$$\left[F_A^{x.e}\right]^n(\forall\alpha.\alpha) \;=\; \sqcap\{\sigma :\; A \vdash \textit{let}\ x{=}\bot\ \textit{in}\ \underbrace{\textit{let}\ x{=}e\ \textit{in}\ \ldots}_{n\ times}\ \textit{in}\ x{:}\ \sigma\}$$

Proof:

Straightforward induction on n using pricipality.

(†): Adding $A \vdash e{:}err$ as an axiom simplifies the formalism in some places.

6. Type Assignment

Following /DM/ we define an algorithm (here semi-algorithm since we do not guarantee termination but see section 7) which given a type assignment A and a term e produces a substitution S and a type τ such that $SA \vdash e:\tau$. The produced S and τ are in some sense the most general such pair. If there is no such S and τ the program fails or loops.

Recall the definition of $\overline{X}(\sigma)$ from section 3. If A is a type assignment we will write $\overline{A}(\sigma)$ to mean $\overline{X}(\sigma)$ where X is the set of free type variables of A. Recall also the existence of a unification algorithm:

Proposition /Ro/:

There is an algorithm U: Type x Type $\rightarrow$ Subst + {*fail*} such that

(i) If $U(\tau_1,\tau_2) =$ *fail* then there is no substitution S with $S\tau_1 = S\tau_2$.

(ii) If $U(\tau_1,\tau_2) = S$ then $S\tau_1 = S\tau_2$ and any other S' with this property can be factored $S' = RS$ for some substitution R.

Moreover the produced S is idempotent and only acts on variables of τ_1 and τ_2.

We can now define algorithm W, which copies that of /DM/ exactly except for the *fix* case and typographical corrections.

Algorithm W(A,e):

```
case e of
  x:      if $Ax = \forall \alpha_1\cdots\alpha_n.\tau$ then  $(1, \{\beta_1/\alpha_1,\cdots,\beta_n/\alpha_n\}\tau)$ where the $\beta_i$ are new
          else fail                                       and 1 the identity function
  $e_1\ e_2$: let $(S_1,\tau_1) = W(A,e_1)$
          let $(S_2,\tau_2) = W(S_1A, e_2)$
          let $V = U(S_2\tau_1, \tau_2\rightarrow\beta)$      where $\beta$ is new
          in $(VS_2S_1, V\beta)$
  $\lambda x.e_1$: let $(S_1,\tau_1) = W(A\{x:\beta\}, e_1)$  where $\beta$ is new
          in $(S_1, S_1\beta\rightarrow\tau_1)$
   *let* $x=e_1$ *in* $e_2$:
          let $(S_1,\tau_1) = W(A,e_1)$
          let $A_1 = (S_1A)\{x: \overline{S_1A}(\tau_1)\}$
          let $(S_2,\tau_2) = W(A_1,e_2)$
          in $(S_2S_1, \tau_2)$
   *fix* $x.e_1$:
          let $\sigma_0 = \forall\beta.\beta$ where $\beta$ is any type variable     (1)
          let $A_0 = A\{x:\sigma_0\}$                                      (2)
          repeat let $(S_{i+1},\tau_{i+1}) = W(A_i,e_1)$ for $i\geq 0$          (3)
                 let $\sigma_{i+1} = \overline{S_{i+1}A_i}(\tau_{i+1})$                       (4)
                 let $A_{i+1} = (S_{i+1}A_i)\{x:\sigma_{i+1}\}$                (5)
          until $S_{i+1}\sigma_i = \sigma_{i+1}$                                   (6)
          in $(S_{i+1}\ldots S_2S_1, \tau_{i+1})$                                (7)
```

Notes:

1. This definition assumes a language like ML /GMW/ in which there are separate fail values which cause (failure) termination of the whole algorithm. We could simulate such values by using explicit injections and tests into a sum type but this complicates the definition for no gain in clarity.
2. The HOPE language /BMS/ requires a type scheme σ to be specified for each top-level definition and hence the *fix* case could be replaced by the code

   ```
   let $A_0$ = $A\{x:\sigma\}$
   let $(S_1,\tau_1)$ = $W(A_0,e_1)$
   if $S_1\sigma = \overline{S_1A_0}(\tau_1)$ then $(S_1,\tau_1)$ else fail
   ```

   which merely checks that the user did supply a fixpoint.
3. If W is implemented in a side-effecting style and the effect of line 4 achieved by side-effecting τ_{i+1} then we must arrange for this to be undone on loop exit (or to use a new generic instance of σ_{i+1} in the result). Similar comments apply to note 2.
4. The definition of the *fix* x.e case is taken from that of the *let* case in that, for any n, W(A, *fix* x.e) defines S_i $(i \leq n)$ and τ_n so that

   $$W(A,\ \mathit{let}\ x{=}\bot\ \mathit{in}\ \underbrace{\mathit{let}\ x{=}e\ \mathit{in}\ \ldots}_{n\ \mathit{times}}\ \mathit{in}\ x) \;=\; (S_n \ldots S_1,\ \tau_n)$$ or both fail to exist.

   This is apparent from the code.

Proposition (Syntactic) soundness and completeness of W for $\vdash$; Given A,e we have

(i) If W(A,e) succeeds with (S,τ) then $SA \vdash e:\tau$

(ii) If for some S',σ we have $S'A \vdash e:\sigma$ then

  (a) W(A,e) succeeds with (S,τ) and

  (b) $S'A = RSA$ and $R(\overline{SA}(\tau)) \sqsubseteq \sigma$ for some substitution R.

Proof:

A fairly convincing proof can be constructed from the equivalence of approximants such as given in note 4 above and *fix* expressions together with proposition 5.2 giving a principal type for such approximants. However, we prefer to give a separate proof of correctness based on the suggested proof by induction on e in /DM/. We accordingly give the *fix* x.e case inductively assuming (i) for e:

Suppose that the *fix* iteration terminates after n steps (otherwise there is nothing to prove. For $0 \leq i \leq n$ we have

$S_{i+1}A_i \vdash e:\tau_{i+1}$ by the induction hypothesis and line (3) of W.

$S_{i+1}A_i \vdash e:\sigma_{i+1}$ by line (4) and GEN steps.

We hence have

$$\sigma_{i+1} \in \{\sigma' : (S_{i+1}A_i) \vdash e:\sigma'\} = \{\sigma' : (S_{i+1}A_i)\{x:S_{i+1}\sigma_i\} \vdash e:\sigma'\}$$

so $\sigma_{i+1} \sqsupseteq \sqcap\{\sigma' : (S_{i+1}A_i)\{x:S_{i+1}\sigma_i\} \vdash e:\sigma'\} = F^{x.e}_{S_{i+1}A_i}(S_{i+1}\sigma_i)$.

By using $\sigma_{n+1} = S_{n+1}\sigma_n$ we have

$\sigma_{n+1} \sqsupseteq F^{x.e}_{S_{n+1}A_n}(\sigma_{n+1}) = F^{x.e}_{S_{n+1}\ldots S_2S_1A}(\sigma_{n+1})$ since the two subscripts to $F^{x.e}$ only differ at x.

By proposition 5.1 characterising pre-fixpoints we thus have

$$S_{n+1}\ldots S_1 A \vdash \mathit{fix}\ x.e: \sigma_{n+1}$$

and we can derive a corresponding formula with σ_{n+1} replaced with τ_{n+1} by INST. Therefore the inductive case is proved with (line 7) $S = S_{n+1}\ldots S_1$ and $\tau = \tau_{n+1}$.

7. Termination Properties

The above arguments about soundness and completeness were only concerned with W succeeding if and only if there is a certain $\vdash$ derivation. They were not concerned with what behaviour W exhibited in failing to give a successful answer. As in the case without *fix* W may fail because unification fails or because a variable does not have a type in the type assumption. But now a new behaviour can occur - one of the type fixpoint iterations may fail to converge. This new case can actually happen: consider the expression *fix* f.λx.f. It gives a σ_n given by $\forall\alpha_0\cdots\alpha_n.\alpha_0\to\alpha_1\to\cdots\to\alpha_n$. Of course, completeness means that the associated F has no non-*err* fixpoint either. As mentioned in the introduction, the work of /MPS/ is concerned with giving such expressions infinite or circular types.

We now turn to the problem of deriving effective termination criteria with which we can predict beforehand whether a given fixpoint iteration will converge. This section is of a much more tentative nature than the previous sections but is included because it illustrates the problems and because it does give an effective termination criterion which however is a little too strong - it faults some programs which have a convergent type iteration. (Perhaps this provides a good reason for adopting a type system like HOPE in which the user has to give the types of all recursive functions thereby avoiding the problems of this section.)

We can see the problem of determining whether an iteration will converge is very like that of the "occur check" in unification which forbids the unification of α with a term containing *a*. Taking the above example, we see that a type which limits the σ_i would need to satisfy the equations:

$$\sigma \leq \tau \quad \text{and} \quad \sigma = \forall\alpha_1\cdots\alpha_n.\tau'\to\tau$$

which is impossible on symbol counting grounds. The problem appears to pose difficulties for unification due to the ≤ inequality since unification is based on equality relations. The problem does not appear to have the flavour of undecidability but an exact characterisation of convergence does not seem very close at hand either.

The partial solution proposed here is to add the following lines of code to W just before the line numbered (1)

```
    let $(S,\tau')$ = $W(A, \lambda x^1\cdots\lambda x^n.e_1')$          (0.1)
    let $(\tau_1'\to\cdots\to\tau_n'\to\tau_0')$ = $\tau'$              (0.2)
    let $V_i$ = $U(\tau_i',\tau_0')$                                    (0.3)
```

where e_1' is the expression derived from e_1 by replacing its n free occurrences of x with the new identifiers $x^1\ldots x^n$. The effect is still to allow x to take on different types at different occurrences in e_1 (but in a slightly more restricted manner as we demonstrate in the example below). Basically, the idea is that the type τ' of $\lambda x^1\cdots\lambda x^n.e'_1$ is then checked (0.3) to ensure that there is a unifier of τ'_i and τ'_0. This serves to fail the call to W (by the side-effect of U) if τ' has a form like $\alpha \to (\beta\to\alpha)$ produced by $\lambda f.\lambda x.f$ from our example *fix* f.λx.f. Note that the unification of τ'_i and τ'_0 is solely performed to check this and any side effect must be undone.

<u>Theorem:</u>

W is now (i) sound (ii) not complete and provided A is closed (iii) total.

<u>Proof sketches:</u>

(i) Since the modification does not enable W to give any answer it did not give before.

(ii) An example is

fix f. *let* g=f *in* ... g(true) ... g(3) ...

This is failed by the modification to W because g is given a type (not a type scheme) due to line (0.1) and so cannot be differently instantiated at its two occurrences. Programs of this form can however be well typed according to $\vdash$ (and hence the old version of W). Note that if completeness is thought to be a vital requirement it could be restored by restricting $\vdash$ by giving a weaker *fix* rule along the lines of

$$\text{FIX}': \quad \frac{A \vdash \lambda x^1\cdots\lambda x^n.e'_1 : \tau_1\to\cdots\to\tau_n\to\tau_0}{A \vdash \mathit{fix}\ x.e_1 : \sigma} \qquad \begin{array}{l}(\text{if } \tau_i \equiv \sigma = \forall\alpha_1\cdots\alpha_k.\tau_0)\\ (\text{and } \alpha_1\ldots\alpha_k \text{ are not free in } A)\end{array}$$

which corresponds to our derived rule for the expression

$$\mathit{fix}\ x.(\lambda x_1\cdots x_n.e'_1)\ x \cdots x$$

used below. FIX' is of intermediate power between our FIX and Milner's FIX" defined in section 5.

(iii) We first show that the iteration $\sigma_{n+1} = F_A^{x.e}(\sigma_n)$ always converges in a finite number of steps (to a type scheme or *err*) subject to the given condition.

We start by noting that *fix* x.e and *fix* x.(λx₁··λxₙ.e') x ··· x have the same semantics and the former can be well typed in type assumption A whenever the latter can (by transforming derivations). Here e' is derived from e as indicated above.

Now let A be an arbitrary type assumption. Associated with the former expression is the type scheme transformation $F_A^{x.e}$ given in section 5. We can similarly define one for the latter. We define

$$G_A^{x.e}(\sigma) = \prod\{\sigma' : A\{x{:}\sigma\} \vdash (\lambda x^1\cdots x^n.e')x\cdots x : \sigma'\}$$

By the above remark on type derivations we have that $F_A^{x.e}(\sigma) \equiv G_A^{x.e}(\sigma)$ and hence if an iteration $(G)^n(\forall\alpha\cdot\alpha)$ converges to a non-*err* value then so does $(F)^n(\forall\alpha\cdot\alpha)$.

In the following we will assume that the free type variables of A are contained in $\{\gamma_1,\gamma_2,\ldots\}$ and that $\{\alpha_i\}$ and $\{\beta_i\}$ are two further disjoint subsets of TVar.

Now, letting $\forall\beta_1\cdots\beta_m.\tau_1\to\cdots\to\tau_n\to\tau_0 = \prod\{\sigma' : A \vdash \lambda x^1\cdots\lambda x^n.e' : \sigma'\}$ be the most general type for the λ-expression and $\sigma = \forall\alpha_1\cdots\alpha_k.\tau$ with $\tau^i = \{\alpha_{(i-1)k+j}/\alpha_j;1\le j\le k\}\tau$, we can write (by the COMP rule)

$$\begin{aligned} G_A^{x.e}(\sigma) &= \forall\alpha_1\cdots\alpha_{nk}\beta_1\cdots\beta_m.U(\tau_1,\tau^1)\cdots U(\tau_n,\tau^n)(\tau_0) \\ &\qquad \text{if this exists and where the unifiers can only instantiate } \{\alpha_i,\beta_j\} \\ &= \mathit{err} \quad \text{otherwise.} \end{aligned}$$

Finally, we show that the existence of V_i with $V_i(\tau_i) = V_i(\tau_0)$ and the V_i not instantiating the $\{\gamma_j\}$ gives a convergence criterion for $G^r(\forall\alpha\cdot\alpha)$ and hence for $F^r(\forall\alpha\cdot\alpha)$. It suffices to show that there is a $\sigma \neq err$ such that $\sigma \sqsupseteq G(\sigma)$ since by monotonicity $G^i(\forall\alpha\cdot\alpha) \sqsubseteq G^i(\sigma) \sqsubseteq \sigma$ and all bounded increasing sequences are eventual constant.

We start with the case n=1. If $V(\tau_1) = V(\tau_0)$ then we may assume that only γ_i are free in $V(\tau_0)$ by using $V' = RV$ if necessary to instantiate any β_i.

Now $G(V(\tau_1)) = \forall\beta_1\cdots\beta_m.U(V(\tau_1),\tau_1)(\tau_0)$

$\sqsubseteq \forall\beta_1\cdots\beta_m.\ V(\tau_0)$ since V unifies $V(\tau_1)$ and τ_1 and is hence less general than $U(V(\tau_1),\tau_1)$.

$= V(\tau_1)$ as no τ_i is free in $V(\tau_0) = V(\tau_1)$.

For the case n>1 we consider

$$G_i(\sigma) = \sigma \sqcup \forall\alpha_1\cdots\alpha_k\beta_1\cdots\beta_m.U(\tau,\tau_i)(\tau_0).$$

Each G_i is monotonic and the mutual pre-fixpoints of the G_i are the pre-fixpoints of G. Moreover $G^i(\forall\alpha\cdot\alpha) \sqsubseteq (G_n\ldots G_1)^i(\forall\alpha\cdot\alpha) \sqsubseteq G^{ni}(\forall\alpha\cdot\alpha)$. Hence the result.

To apply the result to W in the absence of free variables of A (i.e. no enclosing λ-expressions)we merely note that $F^i(\forall\alpha\cdot\alpha)$ is exactly σ_i of the iteration.

8. ML example

The following example which I actually encountered in my rôle of programmer (it occurred in the ML compiler) shows that not all typing problems can be resolved by sorting recursive definitions into 'really' mutually recursive cliques.

In it *list* and *dlist* are isomorphic data structures having operations hd,tl,null, dhd, dtl,dnull giving respective list processing primitives. The code skeleton was:

```
  let rec f(x: structure) = case x of
          ( basecase(y): ...
          | listcase(y): g(y, (hd,tl,null));
          | dlistcase(y): g(y, (dhd,dtl,dnull)))
        and g(x:α, (xhd:α→β, xtl:α→α, xnull:α→bool)) =
          if xnull(x) then () else (f(xhd x); g(xtl x, (xhd,xtl,xnull)))
```

which was (over the larger body of code) a natural programming solution involving parameterising common code. The *fix* rule we suggest can successfully typecheck this.

9. Conclusions

We have extended Milner's polymorphic type scheme to allow more general typing of recursive definitions as required for languages with mutually recursive top level environments as well as some examples in ML itself. We did this for a minimal language, Exp, but the technique should readily extend to a larger set of type constructors.

We have given an algorithm like Milner's, but with a type iteration to determine the type of recursive definitions. A natural question is whether there is an algorithm to do this without iteration or how to find an exact termination criterion. Pragmatically, there may be grounds for restricting the use of this extended algorithm (as it stands) to the top level of definitions only, due the the exponential cost of analysing nested *fix* definitions.

Acknowledgments

Thanks to Sören Holmström for helpful comments and to Chris Wadsworth for the *fix* f.λx.f example. Financial support was provided by Chalmers and the Swedish STU.

References

/BMS/ Burstall, R., MacQueen, D.B. and Sannella, D.T.
HOPE: an experimental applicative language.
Internal report, Dept. of Computer Science, Edinburgh University, 1980.

/DM/ Damas, L. and Milner, R.
Principal type schemes for functional programs.
Proc. 9th ACM Symp. on Principles of programming languages, 1982.

/GMW/ Gordon, M., Milner, R. and Wadsworth, C.
Edinburgh LCF.
Springer-Verlag LNCS 78, 1979.

/Ho/ Holmström, S.
Polymorphic type schemes and concurrent computation in functional languages.
PhD thesis, Dept. of Computer Science, Chalmers TH, S-412 96 Göteborg, 1983.

/Mi/ Milner, R.
A theory of type polymorphism in programming.
Journal of computer and system sciences, 17(3), 1978.

/MO/ Mycroft, A. and O'Keefe. R.A.
A polymorphic type system for Prolog.
To appear in Artificial Intelligence. Preliminary version in DAI research report. Dept. of Artificial Intelligence, Edinburgh University.

/MPS/ MacQueen, D.B.. Plotkin, G.D. and Sethi. R.
An ideal model for recursive polymorphic types.
Proc. 11th ACM Symp. on Principles of programming languages. 1984.

/MS/ MacQueen, D.B. and Sethi, R.
A semantic model of types for applicative languages.
Proc. Aspenäs workshop 1982.

/Ro/ Robinson, J.A.
A machine oriented logic based on the resolution principle.
JACM 12(1), 1965.