Incremental polymorphic type checking with update
(Preliminary version)

Alan Mycroft
Computer Laboratory, Cambridge University
New Museums Site, Pembroke Street
Cambridge CB2 3QG
United Kingdom
E-mail: `Alan.Mycroft@cl.cam.ac.uk`

Abstract

We describe a variant of Milner's ML type inference algorithm which can be used to perform incremental type checking of programs with partially unspecified functions (or predicates in Prolog). This supports modification (*e.g.* for correction) of procedures defined previously and provides for a convenient treatment of top-level mutual recursion including Prolog-style incremental clausal definition.

The system allows us to: define a function of, say, type $\alpha \rightarrow \alpha$, use it in suceeding functions and then modify its definition to a type instance, such as $list(\beta \times int) \rightarrow list(\beta \times int)$, provided that, in the meantime, it has not been used by other functions at an incompatible instance. Undefined procedures can be treated as having type α (or $\alpha \rightarrow \beta$ corresponding to the function $\lambda x.fail$).

This is useful for the case of languages (like HOPE, Miranda, Haskell and Prolog) which require that all top-level procedures are defined mutually recursively — forward references can then be treated as if defined by $\lambda x.fail$ which is then updated when the proper definition is seen. Such languages generally require either top-level mutually recursive phrases to be textually partitioned or require all procedures to have their type declared before their definition or first use — this is because Milner's type inference for recursive definitions is decidable but too weak and because Mycroft's ML+ generalisation is semantically stronger but undecidable in the absence of advance type declarations.

The algorithm presented here allows mutually recursive functions to be type-checked one-by-one (*online*) with or without type declarations and always gives a type at least as general as the offline algorithm for recursive definitions in ML (or its strongly-connected dependency component variant used in Miranda). With type declarations preceding all forward references it exhibits the decidable type system in HOPE and Prolog.

1 Introduction

Consider an ML-like language and the following example

```
let f [] = e_f^1;
let g x = e_g;
let f (a::b) = e_f^2;
```

where we suppose that `f` occurs within e_g but otherwise `f` and `g` do not occur within e_f^1, e_g, e_f^2 . There are (at least) four possible semantics.

1. `f` is defined partially (for only one value), `g` incorporates this definition and then `f` is *superseded by* (redefined as) a different partial definition which hides the original definition for all subsequent uses (but not those in `g`). ML has this behaviour.

2. `f` is defined partially, `g` incorporates this definition and then `f` is *augmented* to make a total function which is visible for subsequent uses.

3. `f` is defined partially, `g` refers to the top-level name `f` (indeed, expression evaluation at this point would invoke the partial definition) and then `f` is superseded by a different partial definition but this redefinition also affects `g`, updating its meaning.

4. f is defined partially, g refers to the top-level name f and then f is augmented to make a total function with this augmentation also affecting g. Prolog has this behaviour.

Nikhil [10] makes a convincing justification of 3 for the purposes of debugging and proposes an implementation based on re-compilation. Choices 3 and 4 also provide a mechanism for implementing, and for type-inference of, top-level mutual recursion — see below.

The above semantic choices factorise into two primitives: firstly the idea of *hiding* (scope shadowing) versus *updating* re-definitions and secondly the choice of the *merging operation* on the values of two definitions. We introduce a syntax $\sqcup$ for the merging operation, so the final line of the example can be written as

$$\texttt{let f = } \lambda\texttt{[]}.e_f^1 \sqcup \lambda\texttt{(a::b)}.e_f^2\texttt{;}$$

We can easily specify semantic choices such as rightmost choice (1 and 3) and Prolog-style clausal merge (2 and 4) via semantics[1] for $\sqcup$, or even other possibilities such as non-deterministic choice. The exact details of $\sqcup$ are irrelevant to the development — see section 2.2.1. For emphasis in examples, we will write == for hiding and := for update. In the absence of recursion the above four cases can all be reflected using == as:

$\texttt{let f == } \lambda\texttt{[]}.e_f^1\texttt{;}$
$\texttt{let g x == } e_g\texttt{;}$
$\texttt{let f == (}\lambda\texttt{[]}.e_f^1\texttt{)} \sqcup \texttt{(}\lambda\texttt{(a::b)}.e_f^2\texttt{);}$
$\texttt{let g x == } e_g\texttt{;}$ (this line only appears for cases 3 and 4).

It is also possible to give denotational semantics capturing the above semantic choices, but the above syntactic transformations better illustrate the intent of this paper.

The justification of this work is based on the following observations. It is possible to implement updating re-definition by recompiling e_g (and recalculating its type) as in the above (Nikhil [10] shows this is also true in the presence of recursion). Moreover, in the above example the effect was that the type of f became typically more instantiated, and thereby (see section 2.3) possibly also the type of g. Furthermore, in typical polymorphic languages (the requirement includes uniform representation of parameterised types) the generated code for a polymorphic procedure is identical to that for any of its instances. *I.e.* a sufficiency condition to avoid recompilation is that the type of a polymorphic procedure at most changes to an instance.[2]

This suggests an implementation of redefinition for the case where the type of an identifier is allowed to change by at most instantiation as:

- Type check and compile each procedure when it is given, but generating indirect calls via a value-cell[3] à la Lisp. Save its type and value in its type- and value-cells but discard its source text.

- When a procedure is re-defined, attempt to update its type cell to reflect its new type, recursively updating type cells of dependent procedures with their new types if possible. If this fails because it (or one of its dependent procedures) has been used at an incompatible instance,[4] report an error. If it succeeds, update the value cell of the re-defined identifier. The value cells of dependent procedures do not need updating.

---

[1] These make most sense on values which are λ- (or other suitable) abstractions.

[2] We conjecture that a converse holds in that for every type and every non-instance type there is an object of which is a member of the former and not the latter.

[3] We can also save a list of all places a procedure is referenced and update these on redefinition. This can be more efficient but is less elegant in some senses.

[4] Consider otherwise
```
let f(x) = x;
let g(y,z) = if f(y) then z else 1;
let f(x) = x+1;
```

We call this process *incremental polymorphic type checking with update* and it is the subject of this paper.

For the rest of this paper, we will treat all re-definitions to be updating re-definitions instead of hiding re-definitions (because the latter have no more effect than consistently using a new name and avoiding further references to the old name). Further, to encompass *inter alia* Prolog and HOPE we will use the word *procedure* instead of function and use the merge operator $\sqcup$ to abstract whether the generic task of *updating* represents *augmenting* or *superseding* a previous definition.

The remainder of this paper consists of the study of how the type system manages the incremental type inference, including how the currently fashionable type inference by analysis of strongly-connected dependency components is subsumed.

1.1 Mutual recursion

Mutually recursive top-levels pose some difficulties in polymorphic languages. These centre about the fact that Milner's type inference system for ML requires each use of a recursively defined identifier *within the definiens of a mutually recursively defined identifier* to be used at the same type, unlike uses outwith this which can be used at an instance type. For example, suppose that we attempt to define `map` in the usual manner (*i.e.* e_1 does not contain `f` or `g`) and then two (presumed mutually recursive) functions `f` or `g` using `map` at two different instances. Treating this naturally as

$$\texttt{let rec } \{ \texttt{ map = } e_1\texttt{; f = } e_2\texttt{; g = } e_3\texttt{; } \} \texttt{ in } e$$

gives a type check error under Milner's regime.

One way to avoid this is to seek a more permissive type discipline. Mycroft's ML+ type system generalises Milner's by allowing mutually recursively defined identifiers to be used at instance types. This works fine in the presence of type declarations for each procedure but turns out to be undecidable for type inference in general. This is the solution adopted by HOPE and Mycroft/O'Keefe Prolog type system. (Miranda [13] can similarly exploit such type information if given but otherwise falls back on the strongly connected component technique given below.) We require each use of a procedure to be a type instance of the declaration; but we also require its declaration to be an instance of (each clause of) its definition.

Another is to use an offline type inference system for top-level. Top-level procedures are collected and then ordered by static dependency (*i.e.* the name of one occurs in the definiens of another) into a DAG of strongly connected components (*i.e.* truly mutually recursive subsets). These components are then processed in (a linearisation of) the DAG order with Milner's algorithm. For example, the above definition is treated as if it were

$$\texttt{let rec } \{ \texttt{ map = } e_1\texttt{; } \} \texttt{ in let rec } \{ \texttt{ f = } e_2\texttt{; g = } e_3\texttt{; } \} \texttt{ in } e.$$

Note that this solution is not optimal in that it does not solve the problem for truly mutually recursive procedures used at different type instances within one-anothers' bodies.[5]

Mutual recursion at top-level can also be treated by the notion of update. For example, assuming e, e' may contain `f` and `g` the code

```
let f x = e;
let g x = e';
```

can be treated as the following updating (re-) definitions (without forward references):

[5] This can be exhibited by certain (simply) recursive procedures over datatypes such as

$$\texttt{datatype t(}\alpha\texttt{) = a of } \alpha \texttt{ | b of t(}\alpha\texttt{) | c of t(t(}\alpha\texttt{)).}$$

```
let f := λx.fail;
let g := λx.fail;
let f := λx.e;
let g := λx.e';
```

This transfers the problem of determining analogues of strongly-connected dependency components into the type system. It turns out that the *parentage relation* of sections 3.2 and 3.3 naturally holds similar information.

1.2 Connection with polymorphic assignment

One novel aspect of this paper is that we allow a previously defined procedure to be updated in a way which allows its type (and those of dependent procedures) to be incrementally modified.

This form of update differs from the systems devised by Damas [1] and Tofte [12] which incorporate polymorphic assignment into ML. Here, we only allow re-definition of top-level procedures by a meta-level directive — there is no internal assignment operation which performs update within the language. The type behaviour of these alternatives differs in that users of a top-level updatable definition can each use different instances of it (even though it may later be updated) whereas the ML extension to allow $\mathit{let}\ x = \mathit{ref}\,[\,]\ \mathit{in}\ e$, or an open top-level variant thereof, requires x to have the *same* type everywhere in e. Therefore, the trick of implementing the above mutual recursion between `f` and `g` as

```
let f = ref λx.fail;
let g = ref λx.fail;
f := λx.e;
g := λx.e';
```

would not subsume the present work.

2 Technical background

This section recapitulates the background literature and algorithms. Tiuryn gives a good survey in [11].

Milner [7] documents the ML polymorphic type inference algorithm. Damas and Milner [2] express the type inference problem in terms of logical inference rules. Mycroft and O'Keefe [9] show how Prolog admits a similar type system. Mycroft [8] exhibits a more general type-inference rule for recursion than Milner and shows it to be sound. Independently, Henglein [3] and Kfoury, Tiuryn and Urzyczyn [4] show that type *inference* with Mycroft's rule for recursion is equivalent to the semi-unification problem which was later shown undecidable by Kfoury, Tiuryn and Urzyczyn [6]. However, note that the type *checking* problem for Mycroft's rule is decidable and was used in both HOPE and the Prolog type system above. We say more about this in section 3.3. The rest of this section details these claims and may be skipped by a reader familiar with the above work.

Although the main part of this paper will be language independent, the following mini-language, *Exp* ranged over by e, is useful to describe the above work:

$$e ::= x \mid \lambda x.e \mid e\ e' \mid \mathit{let}\ x = e\ \mathit{in}\ e' \mid \mathit{fix}\ x.e$$

where x ranges over a set of identifiers.

In examples we will feel free to incorporate other well-known functional language constructs including tuples, pattern matching and definition by mutual recursion (*let rec*) instead of the simple *fix* construct.

2.1 Types and type schemes

A *type*, ranged over by τ, is given by grammar $\tau ::= \alpha \mid \kappa(\tau_1, \ldots, \tau_{n_\kappa})$ where α ranges over a set of *type variables* and κ ranges over a set of *type constructors* with n_κ being the *arity* of κ. Examples will assume κ contains *int* of arity 0 and $\to$ of arity 2.

A *type scheme*, ranged over by σ, is given by the grammar $\sigma ::= \tau \mid \forall\alpha.\sigma$. A type scheme σ is closed if all its type variables appear bound by $\forall$. A *type environment*, ranged over by A, is a mapping from identifiers to type schemes.

Type schemes can be ordered by the (generic) instantiation[6] order $\sqsubseteq$ — here we just define this for *closed* type schemes. We define $\forall\alpha_1 \ldots \alpha_m.\tau \sqsubseteq \forall\beta_1 \ldots \beta_n.\tau'$ iff there is a substitution θ involving only the α_i such that $\theta(\tau) = \tau'$. This order has least element $\forall\alpha.\alpha$ and we can make it into a *cpo* by adjoining a maximal element *err*. This cpo is finitely branching given a finite number of type constuctors. Moreover, each point σ ($\neq$ *err*) satisfies the property that $\{\sigma' \mid \sigma' \sqsubseteq \sigma\}$ is finite. The $\sqcup$ operation on this cpo is (generic) unification, *e.g.*

$$\forall\alpha.\alpha \to \alpha \ \sqcup\ \forall\alpha,\beta.\alpha \to (\beta \to \mathit{int}) \ = \ \forall\alpha.(\alpha \to \mathit{int}) \to (\alpha \to \mathit{int}).$$

2.2 Type inference systems

Milner and Damas give (essentially) the following rules for infering when an expression e has type scheme σ in type environment A, written $A \vdash e : \sigma$:

$$\text{VAR:}\frac{}{A \vdash x : \sigma} \text{ if } x : \sigma \in A$$

$$\text{LAM:}\frac{A[x:\tau] \vdash e : \tau'}{A \vdash \lambda x.e : \tau \to \tau'} \qquad \text{APP:}\frac{A \vdash e : \tau \to \tau' \quad A \vdash e' : \tau}{A \vdash e\ e' : \tau'}$$

$$\text{LET:}\frac{A \vdash e : \sigma \quad A[x:\sigma] \vdash e' : \tau}{A \vdash \mathit{let}\, x = e\ \mathit{in}\, e' : \tau} \qquad \text{FIX:}\frac{A[x:\tau] \vdash e : \tau}{A \vdash \mathit{fix}\, x.e : \tau}$$

$$\text{SPEC:}\frac{A \vdash e : \forall\alpha.\sigma}{A \vdash e : \sigma[\tau/\alpha]} \qquad \text{GEN:}\frac{A \vdash e : \sigma}{A \vdash e : \forall\alpha.\sigma} \text{ if } \alpha \text{ is not free in } A$$

The placement of types (τ) and type-schemes (σ) is critical.

Mycroft showed the following generalisation of Milner's FIX rule was also sound:

$$\text{FIX+:}\frac{A[x:\sigma] \vdash e : \sigma}{A \vdash \mathit{fix}\, x.e : \sigma}$$

but Henglein and Kfoury, Tiuryn and Urzyczyn showed the problem of determining, given A and e, whether such a σ existed was undecidable in general.

When discussing implementation, it is often more direct to consider the equivalent system where rules SPEC and VAR are replaced by the single rule

$$\text{VAR}'\text{:}\frac{}{A \vdash x : \tau'} \quad \text{if } (x : \forall\alpha_1 \ldots \alpha_n.\tau) \in A \text{ and } \tau' = \tau[\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n]$$

This fact is based on the observation that all uses of SPEC can be moved to occur immediately after a use of VAR. Of course, SPEC is still needed in the presence of FIX+, but this is not necessary if *fix* only occurs in a *letrec* construct (or indeed we can adjust FIX+ so SPEC is not needed). Similarly inferences can be arranged so that GEN only occurs at the end of an inference or just before a LET rule. Kfoury, Tiuryn and Urzyczyn [4] give such as system as an alternative set of typing rules.

[6]Note that this is the inverse of Milner's original definition.

2.2.1 The merge operator, $\sqcup$

For any of the above semantics given in the introduction, the syntactic merge operator, $\sqcup$, can be given the type inference rule

$$\text{LUB:}\frac{A \vdash e : \tau \quad A \vdash e' : \tau}{A \vdash e \sqcup e' : \tau}.$$

This corresponds to the ML rule for function definition by cases. It is therefore natural for languages like Prolog and Miranda in which re-definitions augment previous definitions, but is also clearly sound for cases where the first definition is superseded (*i.e.* the effect of $e_1 \sqcup e_2$ is that of e_2). In this latter case, one might suspect that the rule is over-zealous and that

$$\text{LUB}'\text{:}\frac{A \vdash e' : \tau}{A \vdash e \sqcup e' : \tau}.$$

would suffice. However, our requirement that any redefinition of a procedure causes its type to be at most instantiated (in order to reuse its dependents' code) when combined with LUB$'$ again produces LUB.

It is amusing to note that this rule corresponds to a most-general type scheme for $e_1 \sqcup e_2$ given by $\sigma_1 \sqcup \sigma_2$ where σ_i is the most-general type scheme of e_i.

2.3 Principal types and monotonicity

Under the above rules an expression e in a type environment A has a most-general (principal) type scheme σ from which all its types may be obtained by (generic) instantiation.

A related property is that of *monotonicity* — an expression acts as a monotonic function (under the generic instantiation order of section 2.1) from the type schemes of its free identifiers to its most-general type scheme (where we view failure to type-check as a maximal type). This property is important here to guarantee that instantiating the type of an identifier at most instantiates the types of its dependent definitions.

2.4 Iterative type inference

Mycroft [8] showed that a recursive definition d (or mutually recursive set of such definitions) induces a monotonic function $\mathcal{T}_d$ on type schemes using the FIX+ rule. Given the completion to a cpo by adjoining a top element *err* then $\mathcal{T}_d$ is a continuous function.

This gives a natural iterative process to find the most-general type scheme as the least fixpoint of $\mathcal{T}_d$ as $\sqcup \mathcal{T}_d^n \forall\alpha.\alpha$.

Sadly this does not always terminate (due to undecidability). The example $d \equiv rec\{f = \lambda x.f\}$ induces $\mathcal{T}_d(\forall\alpha_1 \ldots \alpha_n.\tau) = \forall\alpha_1 \ldots \alpha_{n+1}.\alpha_{n+1} \to \tau$.

However, if we have a post-fixpoint σ (satifying $\mathcal{T}_d(\sigma) \sqsubseteq \sigma$) then the termination is guaranteed. We use this in section 3.3 to guarantee termination in the case where Milner's FIX rule *would* give a type — the mere existence of such a type means that the iteration is bounded and hence terminates (possibly at a more general type than the bound).

3 Formalism

We consider a system like ML in which top level-phrases are definitions and re-definitions of the form $f_i = e$ possibly interspersed by expressions e to be evaluated. Sections 3.2 and 3.3 look in detail at the non-recursive and recursive cases.

We consider definitions to be numbered $1, 2, \ldots$ by so-called *times*, (re-definitions and expressions do not alter the time). Each definition and re-definition updates a value environment and a corresponding type environment. Definitions *extend* environments at a new identifier and re-definitions merely *alter* the value and type associated with the redefined identifier (and possibly other dependent identifiers). We therefore assume that, at time n, a definition $f_n = e$; or a re-definition $f_i = e$; $(i < n)$ may be made or an expression e may be evaluated. We require that the free identifiers of e in the above are contained in the set $\{f_1, \ldots, f_{n-1}\}$. Mutually recursive (or even simply recursive) definitions such as $f_1 = e_1; f_2 = e_2;$ are treated as the definitions $f_1 = \lambda x.\mathit{fail}; f_2 = \lambda x.\mathit{fail};$ followed by the re-definitions $f_1 = e_1; f_2 = e_2;$ thereby respecting the free-identifier constraint. (Note that Prolog has the effect of an environment with *fail* for every predicate identifier which is then updated as clauses are added, one-by-one, to form the full definition.) The effect of this construction is that, at time n, the environment and type environment are linearly ordered lists of identifiers $\langle f_1, \ldots, f_n \rangle$.

The type associated with each defined identifier will be a closed type scheme: a type scheme with free type variables can always be generalised *via* the GEN rule to a closed type scheme if the type environment is closed — induction completes the argument.

3.1 Non-updating definitions

These will be non-recursive due to the requirements placed on the free identifiers of $f_i = e$ above.

We first recall the common implemention of a normal polymorphic type checker as in a language such as ML. There, type environments associate top-level identifiers with *types* instead of closed type-schemes. Free type variables in these types are treated as implicitly universally quantified (we later further assume them to contain disjoint sets of type variables). This causes generic instantiation to become instantiation and generic unification to become simple unification. When an expression e (for example the right-hand-side of a definition $f_i = e$) is encountered, each free identifier *occurrence* is looked up in the type environment to give a type τ. This type τ is then *cloned*, *i.e.* copied consistently substituting each type variable α_i in τ with a new type variable[7] α_j to yield τ'.[8] Type checking *via* unification takes place as usual, resulting in (a type-checking error or) a *type* τ'' which by construction has type variables disjoint from those in the type environment.

If the expression e is part of a definition $f_i = e$, then the resulting type (τ'') is implicitly generalised by the type environment implementation convention, and so the type environment is augmented by $f_i : \tau''$. Similarly τ'' is available to print the result if e is merely to be evaluated.

3.2 Updating, non-recursive, definitions

We now allow for the possibility of an updating re-definition $f_i = e$ which not only is visible to subsequent users, but also affects uses made in previous definitions. In general this affects the types of all dependent definitions as well as the updated one, but the assumption that the new definition at most instantiates the *type* of its definiendum means that the *code* of dependent definitions continues to be valid.

This section considers the non-recursive case — the free identifiers in a definition *or re-definition* of f_i are further required to be a subset of $\{f_1, \ldots, f_{i-1}\}$. This means that an updating definition of f_i cannot refer to an identifier f_j $(j \geq i)$ even if the re-definition occurs at time $n \geq j \geq i$. Section 3.3 considers the recursive case.

An obvious offline algorithm is to save all the program text, accumulating definitions and re-definitions for each f_i and then type-check them as in ML or Miranda. The merge operator, $\sqcup$, enables a definition and its re-definitions to be treated for type-checking purposes as a single definition. Nikhil explains how to reduce the amount of re-compilation, but his algorithm is still offline in that it requires the text of the whole program to kept.

Our online (incremental) algorithm for this uses a *parentage relation* which maintains information as to where the type of a definition was used so that types of such dependent definitions may be updated when needed. The parentage relation (actually partial function) $\Pi \subseteq \{(\alpha, i, \tau) \mid i \in I\!N\}$

[7] It is convenient, as usual, to arrange an infinite sequence of type variables $\alpha_1, \alpha_2, \ldots$ from which new type variables can be taken by incrementing a counter.

[8] See rule VAR$'$ earlier.

associates each type variable α appearing in the type environment A with the type τ to which its ith clone is instantiated. (A type variable in A is cloned to yield initially another type variable which may become instantiated by unification. For a given identifier, f, all the type variables in $A(f)$ will have the same number of clones, *viz.* the number of occurrences of f in the following text. In particular, type variables appearing in types of (so far) unused definitions have no clones.) Initially, Π is empty. So, after

```
let I x = x;
let f = I I;
```

we have $A = [\mathtt{I} : \alpha \to \alpha, \mathtt{f} : \beta \to \beta]$ as usual but also with parentage relation $\Pi = \{(\alpha, 1, \beta), (\alpha, 2, \beta \to \beta)\}$. This provides the basis for arranging for the type of `f` to be updated when the type of `I` is updated.

Now suppose we have a re-definition $f_i = e$;. We calculate the principal type scheme of e as before represented as an implicitly generalised type, τ say, (section 3.1). As before this has disjoint type variables from those in A. We now unify $A(f_i)$ with τ in the usual manner, but afterwards take special action when a parent type variable has become instantiated. (By construction, before unification the type variables in $A(f_i)$ may have clones, but not those in τ.) Suppose type variable α has become instantiated to τ' but also has clones $\tau_1, \ldots, \tau_k$. The action is to unify each τ_i with a new clone of τ' and then to remove α from the parentage relation (see algorithm below).

This action may cause other type variables (associated with dependent definitions and possibly themselves parents) to become instantiated, which then must be similarly dealt with. However, the process terminates due to the non-recursive requirement since updating the type of f_i can only affect the types of f_j $(i < j \le n)$. If we always choose an instantiated parent type variable associated with the smallest f_i then we only change each top-level type at most once.

3.2.1 Algorithm

We assume a *unify* procedure which unifies two type expressions by side-effecting the instantiation state of its contained type variables in the common ML implementation manner.

```
(* clone i t is used for the ith occurrence of an identifier of type t. *)
fun clone i (κ(t₁, ..., t_nκ)) = κ(clone i t₁, ..., clone i t_nκ)
  | clone i α = if ∃t.(α,i,t) ∈ Π then t
                else let t = newtypevar() in
                         Π := Π∪(α,i,t);
                         t end;

fun unifyandpropagate(oldtype, newtype) =
  ( unify(oldtype, newtype);
    while (∃α instantiated to τ, but with k > 0 children in Π) do
     ( for i=1 to k do
           unify(child_i(α), clone i τ);
       Π := Π − {(α,i,t) | 1 ≤ i ≤ k, t a type}));

fun updatetype(f, newtype, typeenv) =
    unifyandpropagate(typeenv(f), newtype);
```

There are minor implementation details to note, such as the fact that it is helpful when unifying two type variables to instantiate the one which has least children to reduce the number of propagation iterations. Note that the invariant that different top-level types are disjoint is maintained (we exploit the fact, when α and β occur in the same type, that the ith child of α is used at the same occurrence as the ith child of β).

3.2.2 Proof idea

We show that, for one re-definition, the type-environment obtained by this process matches the offline "accumulate all definitions and then typecheck". The result follows for an arbitrary interleaving of definitions and re-definitions by induction. Interspersed expression evaluations may be more generously treated by our online scheme than the offline one (which would defer them all to the end), but these are considered ephemeral.

3.3 Recursion

We now remove the restriction whereby free identifiers in redefinitions of f_i were required to be a subset of $\{f_1, \ldots, f_{i-1}\}$ and hence allow recursion.

Recursion can cause the above algorithm to fail to terminate due to the ability to manufacture (effectively) cyclic parentage relations. Consider applying the above algorithm to the code

```
let f = λx.fail
let f = λx.f
```

Just before the attempted unification of the two types of `f` we have the old and new types of `f` as $\beta_1 \to \alpha_1$ and $\gamma \to (\beta_2 \to \alpha_2)$ with parentage relation $\{(\beta_1, 1, \beta_2), (\alpha_1, 1, \alpha_2)\}$. The problem is that unification of the two types for `f` instantiates α_1 to $(\beta_2 \to \alpha_2)$, which hence unifies α_2 with a new clone of $(\beta_2 \to \alpha_2)$ thereby repeating the process forever. The possibility of such loops is unavoidable due to undecidability without extra contraints on the recursive definitions. We accordingly seek restrictions to avoid such cases.

The introduction notes the problems with the various treatment of mutually recursive top-levels which can be summarised —

1. Milner's FIX rule is over-restrictive.

2. Milner's FIX rule can be improved by an (offline) pre-pass analysing dependency for strongly connected components.

3. Mycroft's FIX+ rule has only semi-decidable most-general type scheme finding.

4. Mycroft's FIX+ rule can be tamed by requiring all procedures to have their type declared before definition or use.

We now explore to what extent we can use our system for the above four purposes.

The incremental type-checking algorithm above always ensures each use of an identifier is a distinct generic instance of its defintion and hence corresponds directly to case 3. Case 4 can be easily simulated too. Given a type-declaration, say `dec` $f : \tau$ before each definition of, or reference, to a given identifier f, we can translate this into a procedure definition (to be updated) as $f = (\lambda x.\mathit{fail}) : \tau$. Moreover, we can treat the type variables in τ as *non-instantiable*, *i.e.* they give an error message on attempting to instantiate them as do type variables used in type constraints in ML. Since this constrains the type of f there is no possibility of looping caused by repeated instantiation of τ. Moreover, it becomes clear that such type declarations are only required before forward references to a procedure (including recursion) and not before procedures which only depend on previous definitions.

A further possibility, which has not been explored by implementation, is to observe when a (parentally-cyclic) update to a type occurs (such as would occur at the beginning of a loop such as given above) and to check whether all the cloned types in the loop admit simultaneous unification with their parents. If so, then the unification gives a post-fixed point to the induced type functional, so the iteration is bounded and will hence terminate because of the finiteness property of the type-scheme cpo (see section 2.4). If not, then we query the user that the type for f appears hard to determine and invite f to be pre-declared (to resolve the possibly undecidable problem).

To help understanding of the relationship to implementation by Milner's FIX rule, consider the following program (where $\perp$ is a non-updatable identifier and hence records of its cloning in the parentage relation are therefore unnecessary) of type $\forall\xi.\xi$:

`let g = λx.fail`
`let f = λx.g ⊥`
`let g = λx.f ⊥`
	corresponding to	`letrec f = λx.g ⊥`
		`   and g = λx.f ⊥`

After this sequence we find the environment is $[\mathtt{f} : \gamma \to \beta', \mathtt{g} : \delta \to \beta]$ with parentage relation $\{(\alpha, 1, \alpha'), (\beta, 1, \beta'), (\beta', 1, \beta), (\gamma, 1, \gamma'')\}$. Note the (harmless) cyclic dependence between β and β'.

3.3.1 Emulating monomorphic FIX

Intuitively, the representation of 1 and 2 within our system seems clear. In Milner's simple *rec* case we simply collapse by unifying all clones with their parents. For the strongly connected dependency component variant we unify every clone (corresponding to a use occurrence) lying on a cycle with that of its parent.

However, this violates the previous requirement that top-level types have disjoint sets of type variables. The above example leads to $\mathtt{f} : \gamma \to \beta, \mathtt{g} : \delta \to \beta]$ and this appears necessary so that any updates to f and g keep in step. The problem for the algorithm is that clonings of β and γ no longer correspond.

Actually, the disjointness requirement can be relaxed to a condition that top-level types have *disjoint or identical* sets of type variables. In general this can be achieved by constructing a type which contains the two types as subtypes. In the above example $(\gamma \to \beta) \times (\delta \to \beta)$ suffices. We then use the relevant subtype *after* cloning which ensures the 1–1 correspondence between sets of clones of type variables appearing in the same type continues to hold.[9]

Subject to this modification it would appear that the above unification of each clone (corresponding to occurrences forming a dependent cycle) with its parent indeed recovers case 2. Note that clones which do not form part of the cycle should not be touched as they represent generic uses of a mutually recursive group of procedures by outsiders. Case 1 is obtained by similarly unifying all clones with their parents, but is practically useless.

3.3.2 Type dependence is better than static dependence

Although the above description was intended to show how we could simulate the strongly connected dependency component type inference algorithm in an incremental manner (using our technique and the observation that strongly connected components are also incrementally computable), we can actually do rather better.

The only way in which our incremental type inference algorithm can loop is if there is a cycle of *type dependence*. (This is less restrictive the strongly connected static dependency component description above, since it still allows us to disregard mutually recursive cycles of procedures with no cyclic type dependence, for example if the cycle contains a non-polymorphic procedure.)

In principle type dependence implies static dependence, but we have to be a little careful how this is stated, if we have used unification on other cycles which cause types to contain non-disjoint sets of type variables.

Acknowledgment

This research was supported by SERC grant GR/H14465.

[9] Perhaps a better data-structure for the parentage relation exists in which this happens automatically.

References

[1] Damas, L. "Type assignment in programming languages", Ph.D. thesis, Edinburgh University, 1985. Available as Computer Science Report CST-33-85.

[2] Damas L. and Milner, R. "Principal type schemes for functional programs", Proc. ACM Symp. on Principles of Programming Languages, 1982.

[3] Henglein F., "Type inference and semi-unification", Proc. ACM Symp. on Lisp and Functional Programming, 1988.

[4] Kfoury, A.J., Tiuryn, J. and Urzyczyn, P. "A proper extension of ML with an effective type discipline", Proc. ACM Symp. on Principles of Programming Languages, 1988.

[5] Kfoury, A.J., Tiuryn, J. and Urzyczyn, P. "On the computational power of univerally polymorphic recursion" Proc. IEEE Symp. on Logic in Computer Science, 1988.

[6] Kfoury, A.J., Tiuryn, J. and Urzyczyn, P. "The undecidability of the semi-unification problem", Proc. ACM Symp. on Theory of Computing, 1990.

[7] Milner, R. "A theory of polymorphism in programming", *JCSS* 1978.

[8] Mycroft, A. "Polymorphic type schemes and recursive definition", Proc. Int. Symp. on Programming, Springer-Verlag LNCS vol. 167, 1984.

[9] Mycroft, A. and O'Keefe, R. "A polymorphic type system for Prolog", *Artificial Intelligence*, 1984.

[10] Nikhil, R.S. "Practical polymorphism", Proc. Functional Languages and Computer Architecture, Springer-Verlag LNCS vol. 201, 1985.

[11] Tiuryn, J. "Type inference problems: a survey", Proc. Math. Foundations of Computer Science, Springer-Verlag LNCS vol. 452, 1990.

[12] Tofte, M. "Operational semantics and polymorphic type inference", Ph.D. thesis, Edinburgh University, 1988. Available as Computer Science Report CST-52-88.

[13] Turner, D.A. "Miranda: a non-strict functional language with polymorphic types" Proc. Functional Languages and Ccomputer Architecture, Springer-Verlag LNCS vol. 201, 1985.