Polymorphism and Unification of Cyclic Terms

Martin Elsman
mael@cs.berkeley.edu
Computer Science Division, University of California, Berkeley

July 7, 1999

Abstract

In this note, we describe an implementation of unification of cyclic terms, called uterms. The implementation allows for ML-style polymorphism, that is, uterms containing free variables can be quantified to form so called uterm schemes, and uterm schemes can then be instantiated (i.e., copied, with fresh variables substituted for quantified variables) for each use. For efficiency, the implementation of polymorphism builds on a notion of variable levels, which are used to guarantee that those variables that are quantified do not occur outside of the uterm being quantified. The note also describes how to decide if a uterm scheme is an instance of another uterm scheme, and we provide an extension to explicit polymorphism, which supports polymorphic recursion and polymorphic analyses of large programs.

The implementation is used for a version of qualifier inference for C, based on unification, which allows for C declarations and C definitions to be specified to be polymorphic in qualifiers.

1 Introduction

- unification [Rob65], unification of cyclic terms [ASU86], unification using levels [Rém92]
- qualifier inference [FFA99], why are cyclic terms necessary (recursive C structs)
- outline of note—gradual refinement of variables

2 Union-Find Terms

The terms that we shall define are based on an underlying polymorphic union-find structure

\[
U : \text{sig type 'a elem}
\]
\[
\begin{align*}
\text{val eq : 'a elem * 'a elem -> bool} \\
\text{val mk : 'a -> 'a elem} \\
\text{val get : 'a elem -> 'a}
\end{align*}
\]
val find : 'a elem -> 'a elem
val union : ('a * 'a -> 'a) -> 'a elem * 'a elem -> unit

end

which allows for creation of cyclic terms. The elements of the union-find structure can be
compared for equality and there are functions mk and get for constructing an element from
the information to be attached to the element and for retrieving the information attached to
an element. Two elements can be unioned using the union function that takes—in addition
to the two elements that are unioned—a function for joining the information attached to the
two elements. After two elements e1 and e2 are unioned, the expression

U.eq(U.find e1, U.find e2)

evaluates to the value true.

We now define a notion of terms on top of the union-find structure. The terms that we
shall define are called uterm. The data structure of terms is defined by the following types:

datatype term = VAR of var
  | CONS of con * uterm * uterm

withtype uterm = {term:term, mark:bool ref} U.elem

and var = {id:int}

and con = string

The information attached to a uterm denotes either a variable or a constructed term. The
information also holds a mark, which can be used to guarantee termination when a uterm is
traversed.

It is straightforward to extend the implementation to records and constructors with any
finite arity.

We shall assume a function fresh: unit -> uterm that creates a uterm containing a
fresh variable and a fresh mark initially set to false.

3 Unification

Unification of two uterm is based on the unification algorithm found in [ASU86, Chapter
??]. The main technicality of the algorithm compared to unification algorithms for non-
cyclic terms is how constructed terms are unified: To ensure termination of the algorithm,
the nodes representing the two constructed terms are unioned before the children of the
constructed terms are unified.

fun get_term (t:uterm) = #term(U.get t)

exception Unify

fun combine p (i : {term: term, mark: bool ref}, i') =
  case (#term i, #term i')
    of (VAR v, VAR v') => if p v then
     if p v' then raise Unify else i

else i'
| (VAR v, _) => if p v then raise Unify else i'
| (_, VAR v') => if p v' then raise Unify else i
| _ => raise Unify

fun restr_unify p (t, t') =
   let val t = U.find t
      val t' = U.find t'
   in if U.eq(t, t') then ()
      else case (get_term t, get_term t')
        of (CONS(c,t1,t2), CONS(c',t1',t2')) =>
            if c = c' then (U.union #1 (t, t'); (* to terminate *)
               restr_unify p (t1, t1')
            restr_unify p (t2, t2'))
      else raise Unify
   end

fun unify0 (t,t') = restr_unify (fn _ => false) (t, t')
fun unify (t,t') = (unify0 (t, t'); t)

It shall become apparent later (when we discuss how to decide if a uterm is an instance of a polymorphic uterm) why we take the effort of implementing the \texttt{restr\_unify} function, which disallows variables that satisfies the given predicate to be unified with other uterm.

As an example of how to construct a cyclic uterm, a uterm \texttt{trec} representing the solution to the equation \( a = c(a,a) \), where \( a \) is a variable and \( c \) is a constructor, is constructed by the declarations:

\[
\begin{align*}
\text{fun cons}(c,t1,t2) &= \text{U.mk}\{\text{term=CONS}(c,t1,t2),\text{mark=ref false}\} \\
\text{val a} &= \text{fresh}() \\
\text{val t} &= \text{cons}("c", \text{a, a}) \\
\text{val trec} &= \text{unify}(\text{a, t})
\end{align*}
\]

As mentioned earlier, each uterm has associated with it a mark. We assume functions for setting, unsetting, and querying marks:

\[
\begin{align*}
\text{val mark} : \text{uterm} \rightarrow \text{unit} \\
\text{val unmark} : \text{uterm} \rightarrow \text{unit} \\
\text{val is\_marked} : \text{uterm} \rightarrow \text{bool}
\end{align*}
\]

The marks can be used to ensure termination when traversing a cyclic uterm. For instance, a printing function

\[
\begin{align*}
\text{val pr} : \text{uterm} \rightarrow \text{string}
\end{align*}
\]

can be implemented, using marks, so that \texttt{pr(trec)} gives the string \texttt{"c(\#,\#)"}. Other uses of the marks include instantiation of polymorphic uterms (Section 4) and computation of free variables (Section 5).
4 Uterm Schemes and Instantiation

A uterm scheme is a pair \((\text{bvs}, \text{t})\) of a possibly empty list of bound variables \(\text{bvs}\) and a body, the uterm \(\text{t}\):

\[
\text{type uscheme} = \text{var list} * \text{uterm}
\]

For implementing instantiation of polymorphic cyclic uterms, it is helpful to extend variables (see Section 2) with a field for instantiation. At the same time we also add a field for marking of variables, which comes in handy for finding free variables of uterms and uterm schemes. We note that this new mark, which is called a \(\text{vmark}\), is different from the marks on uterms. The type \(\text{var}\) is thus refined as follows:

\[
\text{and var} = \{\text{id: int, inst: uterm option ref, vmark: bool ref}\}
\]

The function \(\text{fresh}\) also needs to be refined. It now returns a uterm with a fresh \(\text{mark}\) field, and a fresh variable with a fresh \(\text{inst}\) field, initialized to \(\text{NONE}\), and a fresh \(\text{vmark}\) field.

In the following, we assume a structure \(\text{M}\) that provides functionality for associations from uterms to uterms:

\[
\text{M : sig type map}
\]

\[
\text{val empty : map}
\]

\[
\text{val singleton : uterm} * \text{uterm} \rightarrow \text{map}
\]

\[
\text{val plus : map} * \text{map} \rightarrow \text{map}
\]

\[
\text{val lookup : map} \rightarrow \text{uterm} \rightarrow \text{uterm list}
\]

\[
\text{val minus : map} * \text{uterm} \rightarrow \text{map}
\]

The function \(\text{singleton}\) takes a pair of two uterms and returns a singleton association. The function \(\text{lookup}\) returns a list of those uterms that a uterm is associated with, and the function \(\text{minus}\) takes a map and a uterm and returns the map with the uterm removed from the domain. The remaining functionality of the \(\text{M}\) structure is self-explaining. We should note here that an implementation of the \(\text{M}\) structure has to be based on association lists and use of equality to find associations in a list. Because the maps that we shall encounter tends to be small, this inefficient representation does not slow down the implementation, dramatically.

Instantiation uses a helper function \(\text{copy}\) to copy the body of the uterm scheme and to substitute uterms for the bound variables. The \(\text{copy}\) function takes as argument a uterm, and a list of term pairs that must be unified. The function returns a triple of a uterm \(\text{t}\), a mapping from uterms (pointers) to uterms variables, and a list of uterm pairs, each of which need be unified so as for the uterm \(\text{t}\) to be a correct copy of the original. The reason for delaying the unification of uterms is that we later shall refine unification to use marks, and the \(\text{copy}\) function uses marks as well; we want to be sure that the two uses of marks do not conflict. Here is the definition of the \(\text{copy}\) function:

\[
\text{fun copy (t, T) =}
\]

\[
(* T : delayed unification *)
\]

\[
\text{let val t = U.find t}
\]
in case get_term t
  of VAR {inst=ref(SOME t'), ...} => (t', M.empty, T)
  | VAR {inst=ref NONE, ...} => (t, M.empty, T)
  | CONS(c,t1,t2) =>
    if is_marked t then let val a = fresh()
      in (a, M.singleton(t,a), T)
    else
      (mark t;
       let val (t1', m1, T) = copy (t1, T)
       val (t2', m2, T) = copy (t2, T)
       val m = M.plus(m1,m2)
       val t' = cons(c,t1',t2')
       val T = fold1 (fn (t, T) => (t,t')::T) T (M.lookup m t)
      in unmark t;
       (t', M.minus(m,t), T)
      end)
end

Notice that the marking that takes place makes sure that copying of cyclic terms are done correctly, although the necessary unification is delayed. Instantiation of a uterm scheme is now defined by a function instance with type uscheme -> uterm:

fun instance ((vars, t) : uscheme) =
  (app (fn v => #inst v := SOME(fresh()))) vars;
  let val (t', _, T) = copy (t, [])
  in app unify0 T;
   app (fn v => #inst v := NONE) vars;
   t'
  end)

The present implementation of copy is not good at preserving sharing. For instance, consider the result of instantiating the uterm scheme s:

val a = fresh()
val t = cons("c1",a,a)
val s : uscheme = ([], cons("c2",t,t))
val t_inst = instance s

Then the sub-terms of the c2 constructor in t_inst do not share! One can modify the implementation of copy, so that only those sub-terms that contain instantiated nodes are copied.

5 Finding Free Variables

In this section, we describe how to find free variables of uterms and uterm schemes. We first assume functions for setting, unseting, and querying vmasks:
val vmark : var -> unit
val vunmark : var -> unit
val is_vmarked : var -> bool

We can now define a helper function for finding the free variables of a uterm. The function uses an additional parameter acc for accumulating free variables. It also takes as argument a predicate function on variables:

fun fv0 p (t, acc) =  
  let val t = U.find t  
  in if is_marked t then acc (* for termination *)  
     else case get_term t  
        of VAR v => if is_vmarked v orelse not(p v) then acc  
           else (vmark v; v::acc)  
           | CONS(_,t1,t2) => (mark t;  
             fv0 p (t1, fv0 p (t2,acc))  
             before unmark t)  
     end

Only variables for which the predicate returns true are collected. Here is how to find the free variables of a uterm:

fun fv t = let val vs = fv0 (fn _ => true) (t,[])  
  in app vunmark vs; vs  
end

And here is how to find the free variables of a uterm scheme:

fun fv' (bvs, t) = (app vmark bvs;  
  fv t before  
  app vunmark bvs)

In Section 7.2, we shall use the predicate given to fv0 to limit what variables are accumulated.

6 The Instance-Of Relation

We shall now see why we took the effort of defining the function restr_unify. We assume a function member: 'a list -> 'a list -> bool, which checks if an element is in a list, using Standard ML’s generic equality function. A function to decide if a uterm t is an instance of a uterm scheme s can be defined as follows:

fun is_instance(s, t) =  
  let val vs = fv t @ fv' s  
  val t' = instance s  
  in restr_unify (member vs) (t,t'); true  
end handle Unify => false
7 Controlling Generalisation Using Levels

The machinery that we shall now describe makes it possible to implement type checking and type inference efficiently. The issue that we are confronting has to do with the forming of uterm schemes. As an example, consider the case of ML type inference (algorithm W) [Mil78]. ML type inference is the task of assigning a type to an ML expression. Type inference can be implemented by a recursive function for traversing ML expressions. The function takes as argument a type environment, which maps identifiers (ranged over by id) to type schemes (i.e., uterm schemes) and returns a type (i.e., a uterm.)

To find out which variables in a uterm can be quantified, variables are extended to have an associated level (i.e., an integer); when a variable a with level l is unified with a uterm t then the level of each of the variables in t that have level higher than l are lowered to have level equal to l. During type inference, a current level is maintained. When traversing an expression exp0 of the form

\[
\text{fn id} \Rightarrow \text{exp}
\]

in an environment E then id is bound to a fresh variable a with its level set to the current level. If t is the result of traversing exp in the environment E + \{id \mapsto a\} then the result of traversing exp0 in E is the uterm a \rightarrow t, where \rightarrow is a binary uterm constructor.

When traversing an expression exp0 of the form

\[
\text{let id} = \text{exp1 in exp2 end}
\]

in an environment E, the current level is increased by one before the expression exp1 is traversed and decreased again when returning from the traversal of exp1. Now, all the variables that occur free in the uterm t that is inferred for exp1 and that have level greater than the current level can be quantified (for dealing with side effects, exp1 must also be a syntactic value for any variable to be quantified.\(^1\)) Let s be the uterm scheme formed this way. The uterm resulting from traversing exp0 in the environment E is then the uterm resulting from traversing exp2 in the environment E + \{id \mapsto s\}.

\(^1\)The level of a variable that could be quantified because its level is greater than the current level, but is not, must be lowered to the current level.
# 7.1 Refining Unification

We assume a variable `current_level` for holding the current level and functions `incr_level` and `decr_level` for increasing and decreasing the current level:

```ocaml
val current_level : int ref
val incr_level : unit -> unit
val decr_level : unit -> unit
```

Second, we refine variables to include a `level` field:

```ocaml
and var = {id: int, inst: uterm option ref, vmark: bool ref,
level: int ref}
```

The `fresh` function is refined to create a fresh variable (as before) with the `level` field initially set to the current level:

```ocaml
val fresh =
  let val c = ref 0
  in fn () => (c := !c + 1;
    U.mk {term=VAR {id= !c, inst=ref NONE, vmark=ref false,
    level=ref (!current_level)},
    mark=ref false})
  end
```

We then refine the definition of the `combine` function of Section 3, which is used by the unification algorithm. First we define the functions `lower` and `lower_vars`:

```ocaml
fun level (v:var) = !(#level v)

fun lower (l:int) {term, mark} : unit =
  if !mark then ()
  else (mark := true;
    case term
    of VAR v => if level v > l then #level v := l
      else ()
    | CONS(_,t1,t2) => (lower l (U.get (U.find t1));
                        lower 1 (U.get (U.find t2)))
    mark := false)

fun lower_vars(v1, v2) =
  if level v1 < level v2 then #level v2 := level v1
  else #level v1 := level v2
```

We then refine the function `combine` to be defined as
Here we see the reason why the unification is delayed in the copy function in Section 4: Because the function combine now uses marks, it follows that the function unify uses marks. Thus, one needs to be careful that this use of marks do not conflict with the use of marks in the copy function proper.

7.2 Generalisation

As described earlier, we can form a uterm scheme from a uterm by quantifying all variables in the uterm that have level greater than the current level.

Here is a function for quantifying all variables in a uterm with level greater than the current level:

```haskell
fun quantify_all t =
  let val bvs = fv0 (fn v => level v > !current_level) (t, [])
  in app (fn v => #level v := ~1) bvs;
  (bvs, t)
end
```

So as to be able to distinguish quantified variables from other variables, the level of each quantified variable is set to ~1. The predicate that is passed to the function fv0 (see Section 5) limits what variables are accumulated.

In the next section we shall see that it is sometimes appropriate to limit even further what variables are quantified.

8 Yet a Refinement: Explicit Variables

To provide support for type systems that make use of explicit (type-)variables in programs, we now refine variables to be associated with an optional explicit variable (an optional string):

```haskell
and var = {id: int, inst: uterm option ref, vmark: bool ref,
  level: int ref, expl: string option}
```

The string—representing the explicit variable—in the expl field can be used for pretty-printing purposes. The refinement of the unification algorithm uses only whether the expl
field is NONE or not; an explicit variable is allowed to be unified only with non-explicit variables.

The fresh function is refined as follows:

```ml
val fresh0 =
  let val c = ref 0
  in fn expl => (c := !c + 1;
    U.mk{term=VAR {id= !c, inst=ref NONE, vmark=ref false,
                  level=ref (!current_level), expl=expl},
               mark=ref false})
  end

fun fresh () = fresh0 NONE
```

Instead of providing functionality for generating a fresh variable based on an explicit variable (a string), we maintain a mapping from strings to variables so as to make sure that two explicit variables with the same name and in the same scope are mapped to the same type variable. This mapping from strings to uterms is provided by a structure

```ml
EM : sig val reset : unit -> unit
    val lookup : string -> uterm option
    val insert : string * uterm -> unit
  end
```

The reset function makes it possible for the application programmer to control the scope of explicit variables. We can now provide a function explvar, which takes as argument a string and returns a uterm:

```ml
fun explvar s =
  case EM.lookup s
   of SOME t => t
    | NONE => let val t = fresh0(SOME s)
          in EM.insert(s,t); t
       end
```

To disallow explicit variables to be unified with any other uterm, we first provide a function expl: var -> bool for determining if a variable denotes an explicit variable:

```ml
fun expl (v:var) = #expl v <> NONE
```

We now refine the definition of unify0 as follows:

```ml
fun unify0 (t,t') = restr_unify expl (t,t')
```

We also refine the definition of is_instance:

```ml
fun is_instance(s, t) =
  let val vs = fv t @ fv' s
   val t' = instance s
  in restr_unify (fn v => expl v orelse member vs v) (t,t'); true
  end handle Unify => false
```

10
Finally, we provide a function `quantify_expl : uterm -> uscheme` for forming uterm schemes from uterm schemes by quantifying all explicit variables whose level are greater than the current level:

```ml
fun quantify_expl t =
    let val bvs = fv0 (fn v => (level v > !current_level andalso
        if expl v then true
        else (#level v := !current_level; false)))
        (t, [])
    in app (fn v => #level v := ~1) bvs;
    (bvs, t)
end
```

Here it is important that for those variables that are not quantified, their levels are lowered so that they are not greater than the current level.

### 8.1 On the Size of Terms

Although variables tend to take up more space by each refinement, the space occupied by variables is in many situations very small. The reason is that variables that are unified with other terms immediately become garbage, because the underlying union-find structure associates node information to the equivalent-class-representative, only—as opposed to every node in the graph.

### 9 Type Checking with Explicit Polymorphism

In this section, we shall see how it is possible to use the techniques presented in the previous sections for a form of qualifier inference for C that supports explicit polymorphism in qualifiers. To simplify matters, we assume that a C program is a sequence of function declarations and function definitions, with a definition of a function `main`, which is the entry point of execution. A `function declaration` takes the form

```
dec id : σ ;
```

where σ is a qualifier-polymorphic function-type (implemented as a uterm scheme), and `id` is a function identifier. A `function definition` takes the form

```
def id : σ = exp ;
```

where σ is a qualifier-polymorphic function-type, `id` is an identifier, and `exp` is the body of the function, which may `use` declared or defined function identifiers for function calls. Use of function declarations makes it possible for the programmer to write mutually recursive functions.

Type checking is performed with respect to a `type environment` (TE), which maps function identifiers to pairs of a uterm scheme (implementing the qualifier-polymorphic function-types) and a token, `dec` or `def`, which denotes whether the type scheme stem from a declaration or a definition.
When a function identifier \( id \) is declared with uterm \( \sigma \) and \( id \) does not occur in the type environment, then \( id \) is introduced in the environment with entry \((\sigma, \text{dec})\) and type checking proceeds. If instead the function identifier \( id \) occurs in the environment with entry \((\sigma_0, \text{def})\), then the function \texttt{is_instance}' is used to check if \( \sigma \) is an instance of \( \sigma_0 \)—it is an error if this is not so. Finally, if instead the function identifier occurs in the type environment with entry \((\sigma_0, \text{dec})\) then it is checked that either \( \sigma \) is an instance of \( \sigma_0 \) or \( \sigma_0 \) is an instance of \( \sigma \)—it is an error if either of these properties hold. In the case that \( \sigma \) is an instance of \( \sigma_0 \), type checking proceeds in the current type environment. On the other hand, if \( \sigma_0 \) is an instance of \( \sigma \), then type checking proceeds in the current type environment modified to map \( id \) to the entry \((\sigma, \text{dec})\).

Whenever a function identifier is used in a call to a function, the function identifier is looked up in the environment, and a fresh instance of the uterm scheme is constructed by a call to \texttt{instance}. It is an error if no entry is associated with the function identifier in the environment.

Now, consider the task of type checking a function definition

\[
def \ id \ : \ \sigma = \ exp \ ;
\]

where \( \sigma \) is a uterm scheme \((\text{bvs}, t)\), for some bound variables \( \text{bvs} \) and uterm \( t \). First, we infer a type \( t' \) for \( \exp \). We then make a call to \texttt{restr_unify (member bvs) (t, t')} If the unification fails then an error is reported. Otherwise, if there is already an entry \((\sigma_0, \text{dec})\) for \( id \) in the environment, then it is checked that \( \sigma_0 \) is an instance of \( \sigma \)—it is an error if this is not so. It is also an error if \( id \) occurs in the environment with an entry \((\sigma', \text{def})\), for some type scheme \( \sigma' \). In this way we enforce that a function identifier is defined only once. If no errors occur, type checking proceeds in the current type environment extended to map \( id \) to the entry \((\sigma, \text{def})\).

10 Error Recovery and the Early Unioning

We now address the problem mentioned in Section 6, namely that when unification of two terms fails, the nodes of the possible parents of the two terms have already been unioned in the underlying union-find structure. This unioning, which cannot be regretted, is problematic for visualizing the terms that failed to be unified.

Fortunately, there is a solution to this problem: When two constructed terms are unified, instead of unioning the constructed terms in the underlying union-find structure, we record the equivalence of the two uterm in a list of explicit equivalence classes—represented as lists of uterms—which are then passed around as assumptions to the \texttt{restr_unify} function. Transitivitiy of the equivalence relation is accounted for when new relations are added to the existing explicit equivalence classes. When unification of the children of two constructed terms succeeds, then the two terms are unioned in the underlying union-find structure. Informally, because two constructed terms are unioned after successful unification, the size of the explicit equivalence classes tends to be small.
11 Conclusion

In this note, we have presented an implementation of unification that allows for cyclic terms and polymorphism. The implementation is used for building a version of qualifier inference for C [FFA99] that builds on plain unification and that supports explicit polymorphism. The support for cyclic terms is needed to represent the types of recursive C structs.

References


A The UTERM Signature

In this appendix, we show the interface to an implementation of uterms that is extended to support explicit records and constructed terms of any finite arity.

```ml
signature UTERM =
  sig
  type con = string (* constructor names *)
  type label = string (* labels for records *)
  type evar = string (* explicit variables *)
  type uterm and uscheme (* terms *)

  (* Fresh variables *)
  val fresh : unit -> uterm
  val explvar : evar -> uterm (* look in scope table or *)
  val reset_explvar_scope : unit -> unit (* create new entry *)

  (* Unification *)
exception Unify of string
```
val unify : uterm * uterm -> uterm     (* may raise Unify *)

(* Construction and deconstruction of uterms *)
val cons : con * uterm list -> uterm
val decons : con * uterm -> uterm list option
val decons2 : uterm -> (con * uterm list) option
val record : (label * uterm) list -> uterm
val derecord : uterm -> (label * uterm) list option
val is_var : uterm -> bool
val is_explvar : uterm -> bool

(* Marking of uterms *)
val mark : uterm -> unit
val unmark : uterm -> unit
val is_marked : uterm -> bool

(* Quantification and instantiation *)
val incr_level : unit -> unit     (* Increase current level *)
val decr_level : unit -> unit     (* Decrease current level *)
val quantify_all : uterm -> uscheme     (* Quantify vars with level
   * higher than current *)
val quantify_expl : uterm -> uscheme     (* Quantify explicit vars with
   * level higher than current *)
val instance : uscheme -> uterm     (* Instantiate to fresh vars *)
val is_closed_expl : uscheme -> bool     (* Returns true if no free
   * explicit vars *)

(* Instance-of relations *)
val is_instance : uscheme * uterm -> bool
val is_instance' : uscheme * uscheme -> bool

(* Pretty printing *)
val verbose_printing : bool ref     (* controls printing of vars *)
val pr : uterm -> string
val pr' : uscheme -> string
val pr'' : (uterm -> string) -> uscheme -> string
end