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 : sig type 'a elem
        val eq : 'a elem * 'a elem -> bool
        val mk : 'a -> 'a elem
        val get : 'a elem -> 'a
```

```
        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 *uterms*. 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 uterms 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
             | _ => U.union (combine p) (t, t')
  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 `restr_unify` function, which disallows variables that satisfies the given predicate to be unified with other uterms.

As an example of how to construct a cyclic uterm, a uterm `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:

```
fun cons(c,t1,t2) = U.mk{term=CONS(c,t1,t2),mark=ref false}
val a = fresh()
val t = cons("c", a, a)
val trec = unify(a, t)
```

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

```
val mark : uterm -> unit
val unmark : uterm -> unit
val is_marked : uterm -> bool
```

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

```
val pr : uterm -> string
```

can be implemented, using marks, so that `pr(trec)` gives the string `"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 `(bvs,t)` of a possibly empty list of *bound* variables `bvs` and a *body*, the uterm `t`:

```
type uscheme = var list * 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 *vmark*, is different from the marks on uterms. The type `var` is thus refined as follows:

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

The function `fresh` also needs to be refined. It now returns a uterm with a fresh `mark` field, and a fresh variable with a fresh `inst` field, initialized to `NONE`, and a fresh `vmark` field.

In the following, we assume a structure `M` that provides functionality for associations from uterms to uterms:

```
M : sig type map
        val empty : map
        val singleton : uterm * uterm -> map
        val plus : map * map -> map
        val lookup : map -> uterm -> uterm list
        val minus : map * uterm -> map
    end
```

The function `singleton` takes a pair of two uterms and returns a singleton association. The function `lookup` returns a list of those uterms that a uterm is associated with, and the function minus takes a map and a uterm and returns the map with the uterm removed from the domain. The remaining functionality of the `M` structure is self-explaining. We should note here that an implementation of the `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 `copy` to copy the body of the uterm scheme and to substitute uterms for the bound variables. The `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 `t`, a mapping from uterms (pointers) to uterm variables, and a list of uterm pairs, each of which need be unified so as for the uterm `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 `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 `copy` function:

```
fun copy (t, T) =          (* T : delayed unification *)
  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)
                            end
        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 = foldl (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, unsetting, and querying vmarks:

```
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 -> 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
```

To define a function is_instance' to decide if a uterm scheme is an instance of another uterm scheme, we assume a function

```
val disjoint_vars : var list * var list -> bool
```

which, given two lists of variables, returns true if the two lists are disjoint, and false otherwise. The function is_instance' is then defined as:

```
fun is_instance'(s1, (bvs, t)) =
  is_instance(s1,t) andalso disjoint_vars(fv' s1, bvs)
```

In case one of the two functions returns false, it can be that the function has corrupted the arguments. This possible corruption of the arguments can occur because the unification algorithm unions two constructed uterms before unifying the arguments to the constructors. The error-recovery extension in Section 10 solves this problem.

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

```
fn id => 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 $E + \{\texttt{id} \mapsto a\}$ then the result of traversing exp0 in E is the uterm $a \to t$. where $\to$ is a binary uterm constructor.

When traversing an expression exp0 of the form

```
let id = 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 $E + \{\texttt{id} \mapsto s\}$.

[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:

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

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

```
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:

```
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`:

```
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 l (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

```
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 (lower_vars(v,v'); i)
                          else (lower_vars(v,v'); i')
     | (VAR v, _) => if p v then raise Unify
                     else (lower (level v) i'; i')
     | (_, VAR v') => if p v' then raise Unify
                      else (lower (level v') i; i)
     | _ => raise Unify
```

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:

```
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):

```
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:

```
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

```
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:

```
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:

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

We now refine the definition of unify0 as follows:

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

We also refine the definition of is_instance:

```
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
```

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

```
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

$$\texttt{dec}\ \mathit{id} : \sigma\ ;$$

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

$$\texttt{def}\ \mathit{id} : \sigma\ \texttt{=}\ \mathit{exp}\ ;$$

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 σ and *id* does not occur in the type environment, then *id* is introduced in the environment with entry $(\sigma, \texttt{dec})$ and type checking proceeds. If instead the function identifier *id* occurs in the environment with entry $(\sigma_0, \texttt{def})$, then the function `is_instance'` is used to check if σ is an instance of σ_0—it is an error if this is not so. Finally, if instead the function identifier occurs in the type environment with entry $(\sigma_0, \texttt{dec})$ then it is checked that either σ is an instance of σ_0 or σ_0 is an instance of σ—it is an error if either of these properties hold. In the case that σ is an instance of σ_0, type checking proceeds in the current type environment. On the other hand, if σ_0 is an instance of σ, then type checking proceeds in the current type environment modified to map *id* to the entry $(\sigma, \texttt{dec})$.

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 `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

$$\texttt{def}\ id\ :\ \sigma\ \texttt{=}\ exp\ ;$$

where σ is a uterm scheme `(bvs,t)`, for some bound variables `bvs` and uterm `t`. First, we infer a type `t'` for *exp*. We then make a call to `restr_unify (member bvs) (t,t')`. If the unification fails then an error is reported. Otherwise, if there is already an entry $(\sigma_0, \texttt{dec})$ for *id* in the environment, then it is checked that σ_0 is an instance of σ—it is an error if this is not so. It is also an error if *id* occurs in the environment with an entry $(\sigma', \texttt{def})$, for some type scheme σ'. 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 $(\sigma, \texttt{def})$.

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 uterms in a list of explicit equivalence classes—represented as lists of uterms—which are then passed around as assumptions to the `restr_unify` function. Transitivity 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

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles, Techniques, and Tools.* Addison Wesley, 1986.

[FFA99] Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. A theory of type qualifiers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'99)*, pages 192–203, May 1999.

[Mil78] Robin Milner. A theory of type polymorphism in programming languages. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[Rém92] Didier Rémy. Extension of ML type system with a sorted equational theory on types. Technical report, INRIA-Rocquencourt, October 1992. Rapports de Recherche 1766.

[Rob65] J.A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the Association for Computing Machinery*, 12(1):23–41, 1965.

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.

```
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
```