An Approach to Handling Polymorphic Types in Higher Order Unification

by

Per Kreuger

SICS R88015
Research Report
ISSN 0283-3638

An Approach to Handling Polymorphic Types in Higher Order Unification

by

Per Kreuger
SICS, Swedish Institute of Computer Science
Box 1263, S-164 28 Kista
SWEDEN

Abstract:

The higher order unification procedure as formulated by Huet [Hu 75] unifies terms in the simple theory of types [Ch 40]. In this language types are expressed in a very weak language (no quantification, "->" being the only operator). In many applications a stronger type-system is desirable. Lambda Prolog [MN 86], e.g. uses a type system that is in some ways similar to that of ML. This type-system uses implicitly universally quantified variables in the type-expressions.

It is not trivial to reformulate the higher order unification procedure for such a theory. This paper describes an implementation of the procedure that tries to overcome some of the problems encountered in such an endeavor.

The basic approach taken is to let the types be an integral part of the representation of the terms to be unified. This makes it simpler to instanciate type-variables during the unification process, and to delay the unification of terms with completely unspecified types until such time as more information is gained.

terms. Furthermore we have to modify the unification procedure to handle the instanciation of type-variables, and to delay parts of the unification of two terms if the types of both are variables.

The examples in the following text are all taken from Lambda prolog, as this language is a good example of how the higher order unification can be utilized to implement a higher order language for general programming. It uses polymorphic types but lacks a satisfactory treatment of the case where we try to unify two terms that have variable types. The following section gives a short description of the language.

2. Lambda prolog

Lambda Prolog is a logic programming language based on a generalization of horn-clause logic to higher order logic. It uses a special form or resolution [Nad 87] and a modified version of the higher order unification procedure of Huet [Hu 75]. The syntax of Lambda Prolog permits abstraction and application in the manner of a strongly typed λ-calculus. This gives us the possibility to compute and use a set of functions in a logic programming framework. The set of functions is those we can type in our type-language. In the case of Lambda Prolog this means that we cannot e.g. use recursively defined functions, as these do not have a type expressible in our type-language.

The syntax of application is simply a sequence of terms, i.e. "`f a b`" represents the result of applying the function `f` to its arguments `a` and `b`. Syntactically predicates are just a special case of functions (those returning boolean values), so the call of a predicate looks just like a function application.

The syntax of abstraction uses the \ operator. λx.g(x) is represented as "`X \ (g X)`". There is also a special syntax for lists similar to the one used in standard prolog. "`,`" is used both as a list-element-delimiter and as a logical conjunction. "`;`" is used as logical disjunction.

I have no space here to go further into a presentation of the language itself. I recommend interested readers to get Millers & Nadathurs presentation of the language [MN 86], or one of several other papers they published, describing various applications of the language.

3. Higher order unification

Higher order unification is a procedure that computes a complete set of unifiers (CSU) of two terms e_1 and e_2 . A CSU is a set Σ such that:

$$1)\ \Sigma \subset \{\sigma \mid e_1\sigma = e_2\sigma\}$$

$$2)\ \forall \rho \in \{\sigma \mid e_1\sigma = e_2\sigma\}\ \exists\ \sigma \in \Sigma\ \rho \leq \sigma$$

where the terms e_1 and e_2 are terms in a higher order language (e.g. that of the simple theory of types) , "=" is equality modulo λ-conversion and $\leq$ is defined by

$$\rho \leq \sigma \text{ iff } \exists\eta\ \rho = \eta\sigma.$$

Note that the concept of most general unifier does not apply anymore. The condition 2 above is a weaker variant of the condition on mgu´s. It says that for each unifier ρ of e_1 and e_2 there exists a unifier σ in Σ such that σ is more general than ρ. The the elements of Σ are *not* in general comparable by this ordering.

There does not always exist a CSU where the additional condition,

$$3)\ \forall\sigma_1\sigma_2 \in \Sigma\ \sigma_1 \neq \sigma_2 \rightarrow \neg\ (\sigma_1 \leq \sigma_2),$$

holds. This means there does not always exist a CSU that does not contain redundant unifiers.

The problem of generating such a set of unifiers has long been regarded as a very difficult problem. It is known that in general unification in such a theory is semi-decidable [Hu 73]. This means that if two terms not are unifiable, an algorithm searching for unifiers may never terminate.

There does exist a procedure that will enumerate a set of equivalence-classes of elements of a CSU (if the set is finite) and then either terminate or go into an infinite loop. The equivalence relation is unification of two flexible terms (more on this below). A redundant unifier is always equivalent to a comparable element under this relation. The procedure is due to G. E. Huet [Hu 75].

Huet´s procedure is formulated for the language of the simple theory of types [Ch 40], a typed ω-order language based on the lambda calculus. The basic structure of the unification procedure is similar to the first order case in that it recurs down the

structure of the two terms computing substitutions that make the terms syntactically equal. Lambda conversion introduces additional complexity to the problem of deciding equality, and in Huet´s procedure all terms are kept in a normal form at all times. An additional source of complexity is that the procedure uses two substitution-producing rules. Both the rules themselfs and the choice of which one to use are indeterministic.

Huet distinguishes between rigid and flexible terms. Let

$$\lambda x_1 x_2 \ldots x_n.@(e_1, e_2, \ldots, e_p) \text{ where } n \geq 0 \text{ and } p \geq 0$$

be the normal form of a term. @ is called the head, and $\lambda x_1 x_2 \ldots x_n.@$ the heading of the term. The term is rigid if @ is either a constant of a member of the set

$$\{x_1, x_2, \ldots, x_n\};$$

otherwise it is called flexible. A rigid term keeps its heading unchanged under substitution.

The procedure differentiates between three cases:

1) The heads of both terms are rigid. In this case the heads are simply checked for lambda-convertibility, and the procedure recurs on the bodies of the terms.

2) The heads of both terms are flexible (i.e. not rigid). In this case the procedure does not produce any substitutions that would make the flexible terms equal, but simply assumes that there exists one.

   This is because there always exists a trivial unifier in this case. Moreover if the terms have a function type there always exists an infinite number of unifiers. The effect of the decision to not produce substitutions in this case is that the procedure does not enumerate unifiers, but rather equivalence classes of unifiers as discussed above. The equivalence relation is then the unification of the flexible-flexible pairs that remain after the procedure terminates. This equivalence relation has the properties we desire, i.e. that the comparable elements of the CSU belong to the same equvalence class.

   We have to remember that these flexible-flexible pairs have to represent equal terms, and if the head of one of the terms gets bound later, we have to reinvoke the unification of these terms.

3) One head is flexible and the other is rigid. This is the only case where the

higher order unification proper is utilized. Either of the substitution-producing rules can be used, and it is not trivial which one should be tried first. In some cases the procedure will not terminate if one is tried first, in other cases the situation is reversed. For a description of the substitution-producing rules see [Hu 75].

The unification procedure as formulated by Huet is actually more general than the one sketched above. For example the order in which subterms are chosen for unification is undetermined. This gives more room for heuristics in constructing a search strategy.

4 The problem of polymorphic types

Huet´s procedure operates on terms in the simple theory of types, which is a non-polymorphically typed language. The unification uses the fact that terms are well-typed in this type-system to guarantee that β-reduction will terminate. In addition the substitutions produced depend heavily on the types of the terms. As we have seen Lambda-prolog is a polymorphic language, and I believe that there is good reason for this. Polymorphism relieves the programmer of the burden of writing specialized functions/predicates for each type. The map predicates e.g.

```
mapfun F [X | L] [(F X) | K] :-
   mapfun F L K.
```

would not be very useful as the function variable `F` would have to have a fixed type. You would have to write one such predicate for each type of function you are likely to use. If you allow type-variables, on the other hand, `mapfun` could be assigned the type

$$(A \rightarrow B) \rightarrow (\texttt{list}\ A) \rightarrow (\texttt{list}\ B) \rightarrow o,$$

where `A` and `B` are type-variables and `o` is the constant representing a boolean type. In addition the polymorphic types allow programmers to not specify the types of some of the terms in their programs where this does not effect the semantics of the program. Type inference makes it possible to determine (polymorphic) types of terms that the programmer has not specified.

If we are to use polymorphic types we must modify the unification procedure to handle them. The representation of a term is not complete without an associated variable-free type, but can be seen as a set of terms (namely the set of all terms resulting from instanciating the type-variables to ground types and applying the normalization procedures) in an underlying language. This is an alternative way to look at the polymorphic types. Normally we just say that the term is an instance of a type iff is a

member of the set corresponding to the type.

In many cases the procedure will still work for polymorphically typed terms, but in the case where two terms to be unified has a completely undetermined associated type (i.e. a type-variable) the set of terms in the underlying language represented are infinite, and generate infinite branching in the unification procedure. This corresponds to the case where we have no information about the structure of a term. We only know it is a member of a set of which we, so far, have no information.

The problem occurs in the normalization procedures for higher order terms. The version of the unification procedure that uses η-conversion (the simplest and in most cases the one we want to use) uses the types to convert terms to what Huet calls η–normal form. The η–normal form of a term is computed from its β-normal form by replacing every subterm of the form

$$\lambda u_1 \ldots u_n \cdot @(e_1, \ldots, e_p),$$

where the type of the head @, $\tau(@)$ is

$$\tau(@) = (\alpha_1, \alpha_2, \ldots, \alpha_p, \ldots, \alpha_q \rightarrow \beta) \qquad q \geq p$$

by the term

$$\lambda u_1 \ldots u_n \, w_1 \ldots w_{q-p} \cdot @(e_1, \ldots, e_p, w_1, \ldots, w_{q-p})$$

where the w_i are new distinct variables (flexible terms) of the appropriate type.

I.e. we replace all terms containing applications where the arity of the head is greater than the number of terms in the application with a function of the missing parameters of the application. We apply the original application to the missing parameters, in the form of new lambda bound variables, thereby matching the number of parameters with the arity of the head.

The situation is problematic when the result type of the term to be normalized is a type-variable. We cannot construct the η-normal form of the term without having the type completely determined (i.e. ground). In Miller´s implementation the program either terminates with an error-message when this occurs or makes a deterministic assumption about the result-type of the term. This is clearly not satisfactory.

5. Implementation issues.

This section describes some ideas used in an implementation of a unification procedure that tries to overcome the problems that occur with the introduction of polymorphic types in our language. The implementation itself does not claim to very efficient, but rather suggests a view of how higher order unification can be handled in a polymorphically typed language.

5.1 A partial solution to the problem of polymorphic types

We cannot expect to solve the problem of polymorphic types altogether, as the substitutions produced are dependent on the type of the term, and if the type is undetermined, we do not have sufficient information to produce any substitutions. We could assume that the term is of a certain type and succeed. Then on backtracking we would retract our assumption, and make a new one, and so on. The assumed types would become increasingly complex, and the procedure would not terminate.

The terms together with their associated type in this case represent an infinite set of terms in the underlying language, and we should produce an infinite set of substitutions. It would be desirable if the procedure declined gracefully when this occurs; for example, by generating the unifiers in the infinite sequence, as it does in some other cases where the indeterminism generates infinite branching. Another approach is to delay the unification of the terms. The terms are certainly unifiable, if not, this would be reflected by the inferred types of the terms. This means that this choice should preserve the soundness of the procedure.

If we adopt this strategy it may well happen that the undetermined type of two terms to be unified may be partially or completely determined at a later stage in the unification process. If this occurs we should reinvoke the unification of the terms using this new knowledge, in some cases perhaps even producing a finite set of unifiers.

One way to accomplish this would be to never actually perform the η–normalization, but always represent the terms as β-normalized terms that together with the representation of the associated type represents a set of terms in the underlying language as discussed above. This would allow a modified unification procedure to constrain the sets of terms represented to subsets of these. The subsets would contain only those terms that are actually unifiable using the produced substitutions. Computing the largest common subtype of the types is then accomplished by unifying the types and if we apply the substitutions produced by the higher order unification to the terms we will have an instance of the common subtype.

5.2 Datastructures used in the implementation

As we already mentioned we regard a polymorphically typed term in our language as a set of terms in an underlying language with only ground types. All these terms are then instances of the polymorphic type. We represent the terms as pairs `<T, S>`. `T` is a representation of an untyped β-normalized term in our language, and `S` is the type of this term. The unification then operates on the types as a part of the representation of the term. The representation of a term is not complete without its associated type, as would be the case if we operated on η-normalized terms. I am indebted to [Er 87] for many suggestions in choosing the data structures used in the implementation.

Types are represented either as a prolog variable, a prolog constant or as a prolog-term with the functor `f/2` where the first argument is the base-type (result-type) of the function and the second a d-list of the types of the arguments of the function. For example:

```
f(i, D-D) = i
```

represents the type of an individual constant of the type `i` (a function of arity zero), and

```
f(o [f(i, D2-D2), f(i, D3-D3) | D4]-D4)
```

represents the type of a predicate taking two individuals as arguments and returning a boolean value, i.e. of type `o` (true or false). In general the representation of the type

$$\beta_1 \text{ -> } \beta_2 \text{ -> } \ldots \beta_n \text{ -> } \alpha,$$

$$\mathcal{R}\{\beta_1 \text{ -> } \beta_2 \text{ -> } \ldots \beta_n \text{ -> } \alpha\}$$

is represented as

$$f(\mathcal{R}\{\alpha\},\ [\mathcal{R}\{\beta_1\},\ \mathcal{R}\{\beta_2\},\ \ldots,\ \mathcal{R}\{\beta_n\}\ \mid D]\text{-}D)$$

A normalized term is represented as a pair of a β-normalized term with additional type-information, and the type of the term. i.e.

```
<lambda(Lambda-variables, <Head, HeadType>, Args), Type>
```

where `Lambda-variables` is a (possibly empty) d-list of the variables bound by this abstraction, `Head` is a normalized term, `HeadType` the type of `Head` and `Args` is a

(possibly empty) d-list of normalized terms.

5.3 Modifications of the unification procedure.

The unification procedure has to be modified to handle the terms represented as pairs, and if necessary to compute intersections of types by instanciating the type-variables to make the unification succeed.

This can be achieved by letting the unification procedure perform the η-normalization simultaneously with the recursion down the structure of the terms. The procedure does not expect terms in η-normal from, but examines the types of the terms in order to get maximal information about the term from the given representation.

For example assume we have the following two terms

$$e_1 = f : \alpha_1 \text{ -> } \beta_1 \ (X : \alpha_1)$$

and

$$e_2 = F : \alpha_2 \text{ -> } \beta_2 \ (x : \alpha_2)$$

where f and x are variables and α_1, α_2, β_1, β_2 are type-variables.

The types of these terms are exactly the ones that can be inferred from the structure of the terms only. When we try to unify these terms we will begin by computing their largest common subtype. That is accomplished by unifying β_1 with β_2 with a first order unifier σ. We know then, that the result-type of the terms f and F is identical. We then try to find a substitution for f, that will make the terms lambda-convertible. As we already mentioned there are two basic ways to do this. In this example we will look only at one of them. As the terms f and F both have a function type their η-normal form would have to be of the form:

$$\lambda x_1 \ldots x_k.F(x_1 \ldots x_k) \text{ and } \lambda x_1 \ldots x_k.f(x_1 \ldots x_k),$$

where $k \geq 1$ and k depends on the arity of the result, i.e. $\beta_1\sigma = \beta_2\sigma$. The substitution produced would then be

$$<f, \lambda w_1 \ldots w_k.F(h_1(w_1 \ldots w_k), \ldots, h_k(w_1 \ldots w_k))>,$$

where h_i are new distinct variables.

One such unifier would result for each value of k, i.e. for each assumed $\beta_1\sigma = \beta_2\sigma$. What Miller's current implementation does here is to assume that the $\beta_1\sigma = \beta_2\sigma$ is a primitive type i.e. not a function-type. This gives k the value 1, and a unique solution. It also excludes infinitely many solutions. Another way to handle this case would be to delay the final computation of the term until we have more information on $\beta_1\sigma = \beta_2\sigma$. If this information is not at all available we would still have to make some assumption about the arity of these terms when we report the result of the unification to the user. We would not, however, exclude possible solutions as a consequence of making this assumption to early.

The types of the new variables w_i would also depend on the type of $\beta_1\sigma = \beta_2\sigma$, but we need not make any assumptions about these at this stage in the computation. Of course, when we β-normalize the terms, the type of the argument of f, $\alpha_1\rho = \alpha_2\rho$ will have to be unified with the type of w_1 in order for the terms to unifiable.

We could also ask the user to provide us with a type or let the user opt to see an enumeration of solutions, instead of giving an explicit type. We would never do worse than in Miller's implementation as we could always make the same assumptions as he does, only at a later stage in the computation. And we would never have to make an assumption that is not strictly necessary.

The same kind of considerations would have to be made for x and X when we recur on the subterms. In this example no further information is to be gained, and so the difference from Miller's implementation is not so great. In more complex cases however this approach may produce results when Miller's implementation does not.

It is clear that this kind of manipulation of the types, variables, and arguments of functions requires that we have efficient access to both the beginning and end of the lists representing the arguments, lambda-bound variables and argument-types. The d-lists used in the representation gives us this kind of access. In addition it makes it easy to construct new lists with an append predicate in constant time. The decision to represent the terms as pairs, where the type of a term is an integral part of its representation, saves us the time it takes to η-normalize the terms. It also makes it more natural to suspend the computation of the exact structure of the term until absolutely necessary.

5.4 Conclusions

The ideas presented in this article point to a way of implementing a generalized

References

[Ch 40] Church A. *A Formulation of the Simple Theory of Types.* Journal of symbolic Logic 5 (1940) 56- 68.

[Er 87] Eriksson Lars-Henrik Personal Communication

[Hu 73] Huet G. P. *The undecidability of unification in third order logic.* Information and Control 22(3) (1973) 257-267.

[Hu 75] Huet G. P. *A unification algorithm for typed λcalculus.* Theoretical computer science I (1975) 27-37.

[ML 82] Martin-Löf P. *Computer Programming and Constructive Mathematics* in Proceedings of the 6-th International Congress for Logic, Methodology and Philosophy of Science Hannover 1979, North-Holland Publishing Co. Amsterdam, 1982.

[MN 86] Miller D. A. & Nadathur G. *Higher-Order Logic Programming.* Proceedings of the Third International Logic Programming Conference, Imperial Collage, London, England, July 1986.

[Nad 87] Nadathur G. *A Higher-Order Logic as the Basis for Logic Programming.* Ph.D. Dissertation, University of Pennsylvanina, May 87.