Conditional Answers for Polymorphic Type Inference

M.H. van Emden
University of Victoria
Victoria, Canada

Abstract

J.H. Morris showed that polymorphic type inference can be done by unification. In this paper we show that the unification problem can be automatically generated by resolution and factoring acting on a theory of type inheritance in the form of Horn clauses. The format is a variant of SLD-resolution as used in logic programming. In this variant, programs have an empty least Herbrand model, so that all derivations are “failed” in the conventional sense. Yet conditional answers provide as much and as securely justified information as do the successful answers exclusively used in logic programming.

1 Introduction

In logic programming one is usually interested in successful derivations, which allow one to conclude an affirmative answer to the query. In addition, we have negation by failure as a rule of inference allowing one to conclude the negation of the query. The third alternative occurs when a derivation fails to be successful, not necessarily by failure to find a clause matching the selected goal, but by merely being incomplete. In this case one can extract from the derivation a conditional answer. Vasey has pioneered [10] their use in program transformation, calling them “qualified answers”. This application was taken further in [1], where complete sets of conditional answers are investigated. Note that these uses of conditional answers arise from incomplete rather than failed derivations.
Let us now consider the case of failure. From a finitely failed SLD tree one draws conclusions which are dramatically different, according to whether the closed-world or the open-world assumption is in force. In the first case, the conclusion is a negation. In the second case, when the failure is caused by a lack of information, several alternatives are open.

Sergot's "Query The User" [8] interface for logic programs, reacts to failure to find a program clause matching the selected goal by presenting this goal as a query to the user, in this way regarding the user as an information source providing a potential extension of the program's database.

Cox and Pietrzykowski [2] exploit failed derivations in their use of resolution for diagnosis, where observations are given and their causes are to be determined. When the observation is represented as a query and domain knowledge is given by the program, causes of failure of a resolution derivation correspond to causes of the observations in the problem domain. In this way, valuable information can be extracted from failed derivations.

In this paper we apply conditional answers not to incomplete derivations, as in [10] and [1], but to derivations failed by lack of information. The condition in the conditional answer can only be proved by supplying the lacking information. As a result the conditional answers achieve a greater degree of generality than would be obtained from the corresponding unconditional answer.

An application in which this interesting phenomenon occurs is polymorphic type inference, which we introduce in the next two sections. We then review the concept of a conditional answer yielded by an SLD derivation. In section 5 we show how to construct a derivation by SLD resolution and factoring that yields a polymorphic type inference by means of an answer substitution (as in the usual successful derivation) in combination with the conditions of a conditional answer. In section 6 we present a meta interpreter yielding conditional answers based on SLD resolution with factoring. It solves, as a supplementary benefit, the problem of polymorphic type inference.

2 Types and polymorphism

In logic, types were introduced to avoid antinomies such as the paradox of the Barber. In programming languages they are use-
ful for preventing errors. Both the antinomies and the errors are caused by applying functions to inappropriate arguments. Many such applications can be prevented by allowing a function of a certain class to apply only to arguments of certain classes.

Types are classes of values. For example, the “ceiling” function maps reals to integers. Suppose we call the sets of reals and integers re and int respectively. Then re → int names a set of functions mapping reals to integers. Saying that “ceiling” is of this type means that it belongs to this set. The type expression re → int is an example of one containing only type constants. A type expression such as x → y contains the type variables x and y and can denote any of many different types, depending on the values of x and y.

Expressions with type variables are useful when a function’s type is partly unspecified. For example, the type of the function that takes two functions as arguments and has as value the composition of the arguments, depends on the types of the arguments. But the usual description of composition, \( \lambda f.\lambda g.\lambda x.f(gx) \), is independent of the types of the arguments and therefore describes a function which can belong to any of a class of different types. Thus it is appropriate to associate with a function description such as \( \lambda f.\lambda g.\lambda x.f(gx) \) a type expression containing type variables. Such an expression denotes a class of types; such a class is called a polymorphic type. The polymorphic type inference problem is to determine, given an expression describing a function, an expression for its, possibly polymorphic, type.

For general information about the typed lambda calculus we suggest the text by Hindley and Seldin [3]. For type polymorphism in programming languages, see Milner [6].

3 Morris’s polymorphic type inference algorithm

For the purpose of this paper it suffices to illustrate the action of Morris’s type inference algorithm [7] on an example problem. The algorithm is based on two rules:

1. If \( x : S \) and \( M : T \), then \((\lambda x.M) : (S \rightarrow T)\).
2. If \( M : (S \rightarrow T) \) and \( N : S \), then \((MN) : T\).

Here “\( \alpha : \beta \)” means “\( \alpha \) is of type \( \beta \)".
Figure 1: $\lambda f.\lambda g.\lambda x.f(gx)$ typed by Morris's algorithm.

Let us see how Morris's algorithm uses these rules to determine the type of the composition function $\lambda f.\lambda g.\lambda x.f(gx)$. First, the lambda expression is written in tree form, as in Figure 1.

In the next step, each $\lambda$ variable is assigned a type. As these types are unconstrained, except that occurrences of the same lambda variable\(^1\) must have the same type, the types are assigned in the form of type $\texttt{variables}$. Of course, different lambda variables are assigned different type variables. This does not imply that they must have different types; to do otherwise would constrain them to have the same type.

The two rules can now be used to constrain the types of every unassigned node that has two assigned children. The constraints are written down as the equations in Figure 1.

The equations determine a unification problem: to find a substitution for each variable by a finite term, using $\rightarrow$ as only function symbol, such that for each equation the left and right hand sides become syntactically identical terms.

Morris stated that a lambda expression has a type if and only if the associated unification problem is solvable. A unification algorithm worth the name terminates for every problem indicating either the nonexistence of a solution or having constructed a most general solution unique up to renaming of variables. In this exam-

---

\(^1\)In general, in a lambda expression, different variables may share the same name. Here we assume that such expressions have been transformed so that different lambda variables have different names.
ple there is a solution; the most general one substitutes for \( P \) the type expression \( (T \rightarrow S) \rightarrow ((X \rightarrow T) \rightarrow (X \rightarrow S)) \). Note that \( X, T, \) and \( S \) are type variables.

4 Conditional answers

In logic programming, SLD-derivations (the concept is due to Kowalski [4]; see Lloyd [5] for current terminology) play the role of computations. An attractive property of such computations is that their results are logical consequences of the program regarded as axioms of a logical theory. Let the program be \( P \) and suppose there is a successful SLD-derivation starting with the goal statement \( \leftarrow G_1, \ldots, G_n \). Let \( \theta \) be the composition of the substitutions in the derivation. Then, according to the correctness of resolution,

\[
P \models \forall(G_1, \ldots, G_n)\theta.
\]

Here \( \forall \) indicates that any variables in its argument formula are universally quantified. The right-hand side is called an answer.

For example, let \( \text{App} \) be the program consisting of the clauses

\[
\begin{align*}
\text{app} & \quad ([], Y, Y); \\
\text{app} & \quad ([U|X], Y, [U|Z]) \leftarrow \text{app}(X, Y, Z);
\end{align*}
\]

A successful SLD-derivation exists with \( \leftarrow \text{app}([a, b], X, Y) \) as initial goal statement. Its accumulated substitution is \( Y := [a, b|X] \). Hence the answer is

\[
\forall X. \text{app}([a, b], X, [a, b|X]).
\]

This is a logical consequence of the program \( \text{App} \).

Suppose now the program is \( \text{App}' \), which contains only the second clause of \( \text{App} \). Then no successful derivation exists. Consider the following derivation from \( \text{App}' \):

\[
\begin{align*}
\leftarrow & \quad \text{app}([a, b], X, Y) \\
Y & := [a|Y_1] \\
\leftarrow & \quad \text{app}([b], X, Y_1) \\
Y_1 & := [b|Y_2] \\
\leftarrow & \quad \text{app}([], X, Y_2)
\end{align*}
\]
Although we cannot derive an answer from App', we can derive conditional answers, of which the above derivation furnishes an example:

\[
\text{App'} \models \forall X, Y_2 \cdot \text{app}([a, b], X, [a, b]; Y_2) \leftarrow \text{app}([], X, Y_2)
\]

In general, suppose we have an SLD-derivation from program \( P \) starting with a goal statement containing a single\(^2\) goal \( G \). Suppose that the derivation contains the goal statement \( G_1, \ldots, G_n \) and that \( \theta \) is the accumulated substitution up to this goal statement. Then

\[
P \models \forall (G \theta \leftarrow G_1, \ldots, G_n).
\]

We call the right-hand side a conditional answer.

At first sight, conditional answers appear to be weaker than unconditional ones. In our application of resolution to polymorphic type inference we always obtain conditional answers. We will argue that they are, in this application, stronger, because more general, than an unconditional answer could have been.

## 5 An example of polymorphic type inference by resolution

In section 2 we used two rules for assigning types to lambda expressions. As a first step in the use of resolution for type inference, we formalise these rules as Horn clauses. A prerequisite is a representation of lambda expressions as terms of logic. To do this, we take into account the following considerations:

1. A lambda variable is not a logic variable. Logic is used as a meta language to describe the object language of lambda expressions and their types. Hence, a lambda variable is named by a constant of logic.

2. An abstraction has two components. According to the pragmatics of logic programming, such an object is represented by a logic term with a top-level functor of two arguments. We use \( \lambda \) for this functor. Later, when we switch to Prolog, this functor is shown as an infix dot.

\(^2\)In the application of conditional answers to program transformation this is an important restriction, as the conditional answer becomes part of a logic program and therefore has to be a Horn clause. In the present application the restriction is not needed, but does not get in the way either.
3. An application has two components. For the same reason, it is represented by a logic term having a top-level functor of two arguments. We use \( \circ \), written infix. Later, when we switch to Prolog, this functor is shown as an infix "@".

For example, the lambda expression \( \lambda f.\lambda g.\lambda x.f(gx) \) is named by the logic term

\[
\lambda (f, \lambda (g, \lambda (x, f \circ (g \circ x)))).
\]

We adhere to the usual conventions in logic programming, requiring names of variables to start with a capital letter. Thus, in the above logic term, \( f, g, \) and \( x \) are constants (naming lambda variables).

The rule

if \( x : S \) and \( M : T \), then \((\lambda x.M) : (S \rightarrow T)\)

translates to the Horn clause

\[
\lambda (X, M) : (S \rightarrow T) \leftarrow X : S, M : T.
\]

Note that \( X, M, S, \) and \( T \) are variables and therefore implicitly universally quantified. The rule does not require \( X \) to be a lambda variable, so it also types terms that do not name lambda expressions. This generality is unusual, but not harmful for the purposes of this paper. Note that \( : \) is an infix predicate symbol and that \( \rightarrow \) is an infix functor.

The rule

If \( M : (S \rightarrow T) \) and \( N : S \), then \((MN) : T\).

is represented by the clause

\[
(M \circ N) : T \leftarrow M : (S \rightarrow T), N : S.
\]

Apart from these rules, we will also need a rule for equality. Hence we will use the logic program:

\[
\begin{align*}
\lambda (X, M) : (S \rightarrow T) & \leftarrow X : S, M : T; \\
(M \circ N) : T & \leftarrow M : (S \rightarrow T), N : S; \\
X = X;
\end{align*}
\]

A type inference problem is represented as a goal statement in the sense of logic programming. For example,

\[
\leftarrow \lambda (f, \lambda (g, \lambda (x, f \circ (g \circ x)))) : Z
\]
asks whether a $Z$ exists which is the type of the lambda expression shown.

After five SLD-resolution steps we have:

$$\leftarrow f : S, g : S_1, x : S_2, f : (U \to T_2), g : (V \to U), x : V$$

No goal can now be selected without causing failure. However, whenever two goals are unifiable, the factoring operation is applicable, which leads to the goal statement

$$\leftarrow f : S, g : S_1, x : S_2, S = U \to T_2, S_1 = V \to U, S_2 = V$$

with the term $S \to (S_1 \to (S_2 \to T_2))$ as substitution for $Z$.

The last three goals represent the same unification as the one required according to Morris's method. They succeed and generate the term

$$(U \to T_2) \to ((V \to U) \to (V \to T_2))$$

as substitution for $Z$. The first three goals are unsolvable, hence remain as conditions in the following conditional answer:

$$\forall U, T_2, V.
\lambda(f, \lambda(g, \lambda(x, f \circ (g \circ x)))) : (U \to T_2) \to ((V \to U) \to (V \to T_2)
\leftarrow f : (U \to T_2), g : (V \to U), x : V.$$  

An answer in the usual sense of logic programming only tells us that $Z$, the type required, is

$$(U \to T_2) \to ((V \to U) \to (V \to T_2)).$$

But this information is more useful if it can be related to the constituent types of the function itself. The condition in the conditional answer provides this information. Although we cannot claim that conditional answers are the only way to do so, it seems a surprisingly direct and elegant way.

6 A Prolog program for polymorphic type inference by resolution

The example in the previous section suggests the following algorithm for polymorphic type inference:
• First, let $P$ be the set of clauses

$$
\lambda(X, M) : (S \rightarrow T) \iff X : S, M : T;
\quad
(M \circ N) : T \iff M : (S \rightarrow T), N : S;
\quad
X = X;
\quad
\iff \alpha \ldots \omega : Z
$$

where the last line is the goal statement, in which $\alpha \ldots \omega$ is the logic representation of the term of which the type is to be inferred. From $P$ construct an SLD derivation selecting at each step a goal which does not have a constant as first argument of the predicate "$\ldots\). Such a derivation cannot be extended indefinitely, as each step removes an abstraction or an application operator from the goal statement.

• When a goal statement is reached where all goals have a constant (representing a lambda variable) as first argument of the predicate "$\ldots\), the derivation cannot be continued. At this stage, repeat as many times as possible the following step: if two goals exist in the last goal statement of the derivation of the form $c : t_1$ and $c : t_2$, remove any one of them and add the goal $t_1 = t_2$. Again, this step can only be repeated a finite number of times.

• Finally, continue the derivation selecting equality goals. The original expression has a type if and only if all the equality goals succeed. The type is specified by the conditional answer yielded by the derivation in its final state.

With Prolog available as programming language, it is tempting to use as much as possible of its built-in SLD-resolution mechanism. But there are several differences between the derivation sketched above and the one produced by Prolog:

1. Prolog always selects the leftmost goal.
2. Prolog never factors.
3. Prolog only produces unconditional answers.

These discrepancies suggest a meta-interpreter written in Prolog. We proceed to do this in two stages: in the first stage we write an interpreter giving conditional answers that, in addition, is
flexible in the choice of goal selection strategy. This takes care of the first and third discrepancies noted above. In the second stage, this interpreter is incorporated in the program for type inference, where factoring is provided for.

The meta-interpreter is a small variation on the usual, simplest one exploited to great advantage by Shapiro [9].

%% A meta-interpreter for conditional clauses.

%% The first argument of condAnswer is an atom
%% representing the conditional answer itself; the
%% second argument is a list of the conditions.
%% condAnswer is true if a successful derivation exists
%% using the Prolog selection rule and search strategy
%% and based on a set of clauses given by assertions of
%% the form clause(...)  
condAnswer(Conc,Cond)
  <- select(Conc,Goal,Concl)  % Goal is the selected goal. 
    & cut  % See comment in next clause. 
      & clause([Goal|Body]) 
        % [Goal|Body] is a clause of which the conclusion 
        % matches the selected goal. 
      & append(Body,Concl,Conc2) 
        % Conc2 is the result of replacing in Conc the 
        % selected goal by the body of a matching clause 
        % and applying the matching unifier. 
      & condAnswer(Conc2,Cond); 
condAnswer(Conc,Conc) 
  % <- not(select(Conc,Goal,Concl)); this condition 
  % is optimized away by the cut in the previous 
  % clause. 

append([],Y,Y); 
append([U|X],Y,[U|Z]) <- append(X,Y,Z); 

%% The usual program for appending two lists is 
%% represented by the following two assertions. In 
%% general, each clause of the object language is 
%% represented by a list of atomic formulas of which 
%% the conclusion is the first and of which the 
%% conditions, if any, are the remaining elements of
% the list.
clause([app([],Y,Y)]);  
clause([app([U|X],Y,[U|Z]),app(X,Y,Z)]);

% The first goal of a goal statement is the selected
% goal, as in Prolog.
select([Goal|Goals],Goal,Goals);

Now we change the definition for the relation select to select
any goal with something which is not a lambda variable as first
argument.

% select(X,Y,Z): X is a list of atoms, Y is an atom
% with a term as first argument which is an abstraction
% or an application rather than a lambda variable.
select(Cond,(BVar.Body):Type,Rest)
<- 1Diff(Cond,(BVar.Body):Type,Rest);
select(Cond,(Rator@Rand):Type,Rest)
<- 1Diff(Cond,(Rator@Rand):Type,Rest);

% 1Diff(X,Y,Z): X, Y, and Z are in the list difference
% relation. That is, Z has all the elements of X except
% one, which is Y.
1Diff([X|Xs],X,Xs);
1Diff([X|Xs],U,[X|Ys]) <- 1Diff(Xs,U,Ys);

Next, we change the definition of clause. The previous definition
was just to demonstrate the representation method of clauses
for the meta interpreter by means of a familiar example. Now we
introduce the actual object program to be interpreted by the meta
interpreter. This program consists of the rules for type inheritance.

% S -> T is the type of a function with arguments of
% type S and values of type T. "->" is right-
% associative and has precedence lower than that of ":="
% and higher than those of "." and "@" (see below). The
% use of infix functors is illustrated by the following:
% f.x.f@f@x : (Y -> Y) -> X -> Y if f : Y -> Y and
% x : X where f.x.f@f@x is the lambda term usually
% written as lambda f. lambda x. f(fx). Thus, "@"
% denotes function application. We conform to convention
% by regarding it as left-associative. "." denotes
% abstraction regarded as a function of two arguments:
% the binding variable and the body. We regard it as
% right-associative. The precedence of "." is higher
% than that of "@". E:T iff E has type T. ":" has the
% highest precedence of the infix operators used in
% the program.
clause([X.M : S -> T , X:S , M:T]);
clause([MON : T , M : S -> T , N : S]);

With the new definitions of select and clause, condAnswer
has become a useful building block for the type inference program.
In fact, it does everything we need, apart from factoring.

For factoring we take a shortcut with respect to the algorithm
sketched in the beginning of this section, as it is unnecessary to
unify explicitly. Before factoring, we have a list of goals in which
there may occur pairs which are unifiable. That is, these goals can
be regarded as terms naming the same object. From that point
of view, the list has duplicates in it. So all we need is a program
that produces from a given list a list of the elements, omitting
duplicates.

% factors(X,Y): X and Y are lists of atomic formulas
% and Y is the result of factoring X.
factors(X,Y) <- noDup(X,Y);

% noDup(X,Y): the set of elements of list Y equals the
% set of elements of X and Y has no duplicates.
nDup([],[]);
nDup([X|Xs],Z) <- mem(X,Xs) & cut & noDup(Xs,Z);
nDup([X|Xs],[X|Z])
<- % not(mem(X,Xs)); this condition is optimized away
  % by the cut in the previous clause.
  noDup(Xs,Z);

mem(X,[X|Xs]); mem(U,[X|Xs]) <- mem(U,Xs);

type(Exp,Type,Cond)
 <- condAnswer([[Exp:Type],C) & factors(C,Cond);

% The type of lambda(f,lambda(g,lambda(x,f(gx))))
? type(f.g.x.f@gx),Type,Cond);
Note that we assume again that different lambda variables have different names.

7 Concluding remarks

Initially, we felt towards conditional answers the way the good Dr Johnson felt towards certain speakers\(^3\). Admittedly, we were intrigued by the fact that any information at all could be extracted from derivations that are neither successful, nor failed, but merely incomplete. Still, these answers seemed a weak surrogate compared to the familiar alternatives.

The application to polymorphic type inference shows that such an opinion, if tenable at all, depends on the application. Here, for example, the fact that the type inference is polymorphic, hence more general than other type inference, is contingent on the very conditionality of the answers.

8 Acknowledgments

Many thanks to Michael Levy for helpful discussions; to Rajiv Bagai and the referees for valuable suggestions for improvement of an earlier draft.

The National Science and Engineering Council of Canada has provided generous research facilities in aid of this research.

9 References


\(^3\) "Sir, a woman's preaching is like a dog walking on his hinder legs. It is not done well; but you are surprised to find it done at all."


