A LOGIC PROGRAMMING APPROACH TO MANIPULATING FORMULAS AND PROGRAMS

Dale Miller
Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104

Gopalan Nadathur
Department of Computer Science
Duke University
Durham, NC 27706

18 September 1994

Abstract: First-order Horn clause logic can be extended to a higher-order setting in which function and predicate symbols can be variables and terms are replaced with simply typed λ-terms. For such a logic programming language to be complete in principle, it must incorporate higher-order unification. Although higher-order unification is more complex than usual first-order unification, its availability makes writing certain kinds of programs far more straightforward. In this paper, we present such programs written in a higher-order version of Prolog called λProlog. These programs manipulate structures, such as formulas and programs, which contain abstractions or bound variables. We show how a simple natural deduction theorem prover can be implemented in this language. Similarly we demonstrate how several simple program transformers for a functional programming language can be written in λProlog. These λProlog programs exploit the availability of λ-conversion and higher-order unification to elegantly deal with several of the awkward aspects inherent in these tasks. We argue that writing similar procedures in other languages would yield programs which are much less perspicuous.

Submitted to the 1987 IEEE Symposium on Logic Programming. Comments are welcome.

Address correspondence to Miller at address above or at "dale@linc.cis.upenn.edu".

This work has been supported by NSF AI Center grants NSF-MCS-83-05221, US Army Research office grant ARO-DAA29-84-9-0027, and DARPA N000-14-85-K-0018. The second author has also been supported by a Burroughs contract.

1. Introduction

The data structures that have traditionally been used in a logic programming language such as Prolog have been restricted to first-order terms. It is easy, however, to imagine extending these data-structures to the terms of a λ-calculus, and we believe that there certain gains to be obtained by doing so.

There are several kinds of objects whose representation in a logically correct manner requires a term language that incorporates higher-order notions. Examples of these kinds of objects are provided by programs and formulas. The task of describing the denotations of programs, for instance, requires an allusion to the operations of abstraction and application; it is therefore clear that in order to represent programs in a fashion closely related to their meanings requires the data-structures provided, for instance, by λ-terms. Similarly, an adequate characterisation of the operation of quantification in first-order formulas requires a set of data-structures that provides the notion of abstraction in conjunction with first-order terms.

The use of λ-terms in a logic programming language thus provides us with a facility in representing the above kinds of objects. If function variables were also permitted to be free in λ-terms, then we could use these terms as schemata to represent classes of objects whose meaning have a common "compositional structure," and we could thus use such an extended logic programming language to specify logically meaningful relationships between such classes of objects. For specific examples where such an ability has applications consider the following.

(i) Rules of logical inference can be seen as relationships between formula schemata. Given that such schemata can be represented by the data structures of such a higher-order logic programming language, the process of deduction in a logical system can easily be specified by definite clauses in this language.

(ii) Certain kinds of program transformations [2] can be thought of as relationships between program schemes. Program schemes can be represented by λ-terms in which function variables appear free [7]. Thus definite clauses in this hypothetical language can be used to encode, and thus specify, such program transformations.

The above observations thus reveal a potentially rich realm of applications for a higher-order logic programming language. While the addition of higher-order features to a language like Prolog has been considered in the past, the true potential of such an addition has not really been understood. Some work (e.g. [12]) has been done toward providing higher-order features present in Lisp-like languages through mechanisms for encoding predicate variables. The usefulness of function variables in conjunction with λ-terms has, however, not been recognized. It is common, instead, to dismiss their addition with the observation that higher-order unification is undecidable. While higher-order unification is a complex operation, it is our belief that it provides us a mechanism with which we may solve difficult problems in a conceptually elegant way. Our purpose in this paper is primarily to provide illustrations of the use of λ-terms and of higher-order unification to bolster this claim.

This paper is organized as follows. We describe a higher-order logic programming language called λProlog in the next section. The rest of the paper is devoted to illustrating some of the applications that exist for a language such as λProlog. In section 3 we argue that a logic programming language is a particularly apt vehicle for implementing theorem provers, and we demonstrate how the presence of λ-terms facilitates such an implementation task. The remaining sections show the use of λ-terms in representing programs, and illustrate how transformations between programs may be described through the use of definite clauses. In section 4 we describe λ-term representations for a simple class of functional programs and show how higher-order unification alone can preform non-trivial program transformations among them. In section 5, we extend our representation to functional programs which include a mechanism, popular in various functional programming languages, that employs patterns to specify how a function's arguments are to be decomposed and used. We then present a transformation which can remove such patterns in favor of explicit uses of data type desctructors. Finally, section 6 considers the task of tail-recursion removal to illustrate how the logic of definite clauses may be used to enrich some of the template-matching ideas first presented by Burstall and Darlington [2]. It is to be noted that all the examples discussed in this paper have actually been tested on an implementation of λProlog.

2. The Logic Programming Language λProlog

λProlog is a logic programming language that is based on a higher-order logic which provides predicate and function variables as well as simply typed λ-terms. In this section we provide a brief exposition to this language, since we shall need it in the discussions in the rest of this paper. This exposition is deliberately brief since the theoretical issues pertaining to this language are not of importance in the present context, and have been the object of our study elsewhere [9, 10].

While λProlog contains features such as predicate variables, for the purposes of this paper we ignore these and view this language as one that extends Prolog by replacing first-order terms with typed λ-terms in which function variables may appear. Typed λ-terms are essentially those terms that can be constructed from typed collections of constant and variable symbols via the operations of abstraction and application. Variables are represented in λProlog by tokens with an initial upper-case letter, and all other tokens are constant symbols. The operation of application is represented by writing two terms in juxtaposition, and the operation of abstraction is represented by using the infix symbol `\` between the variable and body of the abstraction. Every term in this language is assumed to be typed. Types associated with variables and constants are either determined through explicit type declarations or are inferred from context. Types associated with complex expressions are obtained in the usual manner: if `X` is a variable of type `A` and `Y` is an expression of type `B`, then `X\Y` is an abstraction of type `A -> B`: likewise, if `F` is a term of type `A -> B` and `X` is a term of type `A`, then `(F X)` is a term of type `B`.

The syntax of definite clauses in λProlog borrows from that of Prolog [1]. The following set of clauses that define two standard predicates on lists illustrates some of these similarities and differences.

```
append nil K K.
append (cons X L) K (cons X M) :- append L K M.
member X (cons X L).
member X (cons Y L) :- member X L.
```

The chief difference here is that it adopts a Lisp-like notation for writing terms, rather than the notation normally used in a first-order language. In these clauses we assume that `cons` and `nil` have the types `A -> (list A) -> (list A)` and `(list A)` respectively (the type constructor `->` associates to the right), where tokens that begin with an upper-case letter stand for type variables; such type variables provide this language with a form of polymorphism. The types of the remaining expressions can be inferred from context given these types. In the examples in this paper, we shall occasionally provide type associations, but in general we shall assume that the reader can infer them from context when it is important.

The notion of equality between two terms in λProlog is richer than that for first-order terms in that it incorporates the notion of λ-conversion. Two terms are thus considered to be equal if they can be interconverted by the rules of α-, β-, and η-conversion. We assume that the reader is familiar with the first two rules, and the last rule asserts that if `F` is a term of functional type in which the variable `X` does not appear, then `F` is equal to `X\(F X)`. It is known that every typed λ-term can be reduced to a normal form by the systematic application of these rules. The question of equality between two closed λProlog terms is, therefore, decidable.

A notion of importance in λProlog is that of higher-order unification, *i.e.* given two, possibly open, λ-terms s and t, we wish to determine if there is a substitution θ for the free variables of s and t such that θs is equal to θt *modulo* λ-conversion. An illustration of this problem is provided by considering the following two λ-terms where we assume that `+` and integers are uninterpreted constants: `X\Y\((G X Y) + 2)` and `X\Y\((g Y X) + H)`. These two terms can be made equal if `G` is instantiated to `X\Y\(g Y X)` and `H` is instantiated to 2. Notice, however, that the following two λ-terms are not unifiable: `X\Y\((G X Y) + (h X))` and `X\Y\((g Y X) + H)`. This is because there is no term which can be substituted for `H` which will make the second summand of the second term depend on its first argument.

Clearly the unification problem in the higher-order setting is more complex than in the first-order setting. In fact, in the general case, determining the existence of higher-order unifiers is undecidable Even when unifiers do exist, there may not be a most general unifier. To see this fact, consider the unification problem posed by the following two terms: `(F a)` and `(g a a)`. There are exactly four different term that could be substituted for `F` that would unify these two terms: `X\(g X X)`, `X\(g a X)`, `X\(g X a)` and `X\(g a a)`. Notice that none of these substitutions subsume another. The non-existence of a most general unifier has the consequence that if the above problem is a part of a more complex problem, then all four of these terms might need to be considered. For example, if the terms `(F b)` and `(g a b)` also need to be unified, then only the second substitution above also unifies the second pair.

Despite the complex nature of higher-order unification, there is a systematic search procedure for determining the existence of unifiers. In fact, Huet in [6] describes a procedure for conducting such a search and for producing a unifier whenever one exists. Based on this procedure an interpreter for λProlog has been implemented. This interpreter mixes unification and backchaining steps in much the same way as Prolog interpreters do. The main difference is that, as we have noted above, higher-order unification can cause branching. The way this is dealt with in λProlog is by using a depth-first paradigm even in the search for unifiers. Although the choice of unifiers may need to be backtracked over, this can be accommodated within the general nature of search in Prolog.

3. Representing and Manipulating Logical Expressions

The term structures of λProlog contain at least one enrichment over first-order terms in that they incorporate the notion of λ-abstraction. This operation is useful whenever there is a desire to represent objects that involve the concept of a variable being bound over the scope of sub-expressions. A situation of this sort arises, for instance, when there is a need to represent first-order formulas as objects that are to be manipulated by programs. This aspect is brought out clearly if we consider the task of representing the formula $\forall x \exists y (P(x, y) \supset Q(y, x))$ as a term in a logic programming language. Fragments of this formula may be encoded into first-order terms, but there is a genuine problem with representing the quantification. We need to represent the variable being quantified as a genuine variable, since logical operations (such as quantifier instantiation) may involve substituting for the variable. A correct representation, however, requires that we distinguish occurrences of the variable within the scope of the quantifier from occurrences outside of it.

The mechanism of λ-abstraction provides the tool needed to make such distinctions. To illustrate this let us consider how the formula above may be encoded using the terms of λProlog. For this purpose we first reserve the sort `b` for the types of terms that represent first-order formulas. Further we assume that the constants `&`, `or` and `=>`, which we shall use to represent the the logical connectives $\wedge$, $\vee$ and $\supset$, are defined to be infix operators of type `b -> b -> b`. Finally, we assume that the constants `all` and `some` are defined to be of type `(i -> b) -> b`; these two constants have the type of "generalized" quantifiers and may be used together with abstraction to represent universal and existential quantification respectively. Assuming these declarations, the formula above may be represented by the λ-term `(all X\(some Y\(p X Y => q Y X)))`.

The ability to represent formulas in a manner that captures all the important logical aspects is of interest because it provides a new domain of application for logic programming languages, namely as a vehicle for implementing proof systems based on natural deduction.

Consider for instance the following rule of inference in a sequent calculus (such as the one described in [4]):

$$\frac{\Gamma \longrightarrow F1 \qquad \Gamma \longrightarrow F2}{\Gamma \longrightarrow F1 \wedge F2}$$

This rule embodies a notion of search that is relevant to the construction of proofs. To be precise, it suggests that one way to construct a proof for the sequent $\Gamma \longrightarrow F1 \wedge F2$ is to construct proofs for the sequents $\Gamma \longrightarrow F1$ and $\Gamma \longrightarrow F2$. Logic programming languages provide us with a mechanism for computation that captures exactly this notion of search. Thus, assuming our representation for formulas and assuming that antecedents of sequents are represented as lists of formulas, the above rule may be described by the following λProlog clause

```
prove Gamma (F1 & F2)  :-  prove Gamma F1, prove Gamma F2.
```

Such a clause may actually be used to search for proofs. Attempting to use it reveals at least one use for higher-order unification; the second-order term $(F1 \& F2)$ would have to be matched with the term that instantiates it in an invocation.

While it may be argued that much of the same advantages may already be derived from first-order term encodings of formulas, a consideration of quantifier rules alters this picture. Take for example the following rule in a sequent style calculus

$$\frac{P(t), \Gamma \longrightarrow F}{\forall x P(x), \Gamma \longrightarrow F}$$

where t is some term. An implementation of this rule requires the instantiation of a quantifier. Given our representation of quantification, this operation may be described rather directly as an application. The intended effect is then achieved by virtue of the rules of λ-conversion. Using this idea, the quantifier rule above may now be easily described:

```
prove (cons (all X\(P X)) Gamma) F :- prove (cons (P T) Gamma) F.
```

Notice that `T` is a logic variable in this definite clause. In the course of constructing a proof, this `T` may be instantiated by an arbitrary term.

To provide a more complete illustration of the usefulness of a language such as λProlog in the context under discussion, let us consider the task of writing an interpreter for the logic programming language that is described in [8] and [3]. This language extends the conventional first-order logic programming language by permitting implications in goal formulas. To be precise, the definite clauses and goal formulas in this language are described by mutual recursion in the following manner; we assume here that A, D, and G

are syntactic variables for (first-order) atomic formulas, definite clauses and goal formulas respectively.

$$D ::= A \mid G \supset A \mid \forall x D, \text{ and}$$
$$G ::= A \mid G_1 \vee G_2 \mid G_1 \wedge G_2 \mid D \supset G \mid \exists x G.$$

A *program* in this language is a finite set of closed definite clauses, and a *query* is a closed goal formula. The relation of being "derived from" between a query G and a program $\mathcal{P}$, denoted by $\mathcal{P} \vdash_O G$, is formalised in [8] in a natural deduction framework by the following proof rules:

(i) $\mathcal{P} \vdash_O \exists x G$ if there is a closed term t such that $\mathcal{P} \vdash_O G[x/t]$.

(ii) $\mathcal{P} \vdash_O G_1 \wedge G_2$ if $\mathcal{P} \vdash_O G_1$ and $\mathcal{P} \vdash_O G_2$.

(iii) $\mathcal{P} \vdash_O G_1 \vee G_2$ if $\mathcal{P} \vdash_O G_1$ or $\mathcal{P} \vdash_O G_2$.

(iv) $\mathcal{P} \vdash_O D \supset G$ if $\mathcal{P} \cup \{D\} \vdash_O G$.

(v) $\mathcal{P} \vdash_O A$ if A is atomic and is an instance of a formula in $\mathcal{P}$.

(vi) $\mathcal{P} \vdash_O A$ if A is atomic and there is an instance $G \supset A$ of a formula in $\mathcal{P}$ such that $\mathcal{P} \vdash_O G$.

These proof rules translate rather directly into clauses in λProlog. Indeed the following list of clauses define the predicate `interpreter` such that the goal `interpreter Clauses Goal` is derivable just in case `Clauses` is a list of terms that represents the formulas in a program $\mathcal{P}$, `Goal` is a term that represents a query G, and $\mathcal{P} \vdash_O G$.

```
interpreter Cl (some G) :- interpreter Cl (G T).
interpreter Cl (G1 & G2) :- interpreter Cl G1, interpreter Cl G2.
interpreter Cl (G1 or G2) :- interpreter Cl G1 ; interpreter Cl G2.
interpreter Cl (D => G) :- interpreter (cons D Cl) G.
interpreter Cl A :- member Clause Cl, instantiate Clause A.
interpreter Cl A :- member Clause Cl, instantiate Clause (G => A),
                    interpreter Cl G.

instantiate (all P) C :- instantiate (P T) C.
instantiate C C.

member X (cons X Rest).
member X (cons Y Rest) :- member X Rest.
```

An interesting aspect of the definite clauses above is the manner in which the predicate `instantiate` is used repeatedly to perform universal instantiations and thus produce a new copy of the body of a clause. It is important to notice that writing an interpreter for the language under consideration really requires an explicit mechanism for variable binding since it is necessary to determine the scope of a quantifier. There is, for instance, a distinction to be made between the two goals $\exists x(\forall y\, p(x,y) \supset q)$ and $(\forall x \forall y\, p(x,y) \supset q)$ since only the latter may be derived from the program $\{p(a,c) \wedge p(b,c) \supset q\}$. First-order terms, of course, do not provide a facility for representing variable binding. It is therefore difficult to see how an interpreter that is equivalent to the one above can be written in Prolog even with the "extra-logical" predicate `clause` [1] that captures some of the behaviour of the predicate `instantiate` above.

4. Programs as typed λ-terms

λProlog is designed to understand the equational nature of simply typed λ-terms. While this provides for very strong data structures, they fall short of fully representing program structures. For example, recursion and data types are missing. Although the latter can be overcome by coding structures such as integers into λ-terms, we shall take a more direct approach to solving both of these deficiencies by introducing constants to help represent recursive programs and data types. The equational nature of these new constants will, of course, not be understood by the unification algorithm, so when these equational properties need to be accounted for, they will be built into λProlog definite clauses.

For the rest of this paper, we shall assume that our programs represent computations over a fixed domain given the type `val`. We shall assume that this domain contains non-negative integers, booleans, list objects, paired objects, and an error object. These data types are arranged in the following fashion.

- We shall assume that there are two booleans, namely `truth` and `false`. There will only be one boolean operation which we need, `and`, which satisfies the obvious equations when applied to two booleans.

- To deal with integers, we have the predicate `intp`, the non-negative integers `0`, `1`, `2`, etc., and the operations `+`, `*`, and `-`. The expression `(intp n)` will be equal to `truth` if `n` is equal to a non-negative integer and equal to `false` otherwise. The meaning of `+`, `*`, `-` will be the standard ones when applied to non-negative integers. (Here, if subtraction yields a negative number, we assume its value is 0.)

- To deal with lists, we have two constructors `cons`, `nil`, the two predicates `consp` and `null`, and the two destructors `car` and `cdr`.

- To deal with pairs, we have one constructor `pair`, the predicate `pairp`, and the two destructors `first` and `second`. Although this data type could be replaced by the list data type, we find it convenient to keep them distinct.

- The constant `error` will denote the error value.

The following two equational formulas, will be implemented directly later.

$$\begin{array}{l} \forall x\, ((\texttt{consp}\ x) = \texttt{truth} \supset x = (\texttt{cons}\ (\texttt{car}\ x)\ (\texttt{cdr}\ x))) \\ \forall x\, ((\texttt{pairp}\ x) = \texttt{truth} \supset x = (\texttt{pair}\ (\texttt{first}\ x)\ (\texttt{second}\ x))) \end{array} \qquad (*)$$

There are many equations between all these structures which would need to be specified to completely explicate the nature of these data types and their interaction. For example, what should `(car nil)` be equal to? None of these issues actually have an impact on the analysis we present below, so do not need to pursue those questions here.

In order to represent simple recursive schemes, we shall introduce the constants `fixpt` and `cond`, of types `(A -> A) -> A` and `val -> A -> A -> A` for all types `A` built using the type `val`. The equations needed for using these constants are simple and given below:

$$\begin{array}{c} \forall x\, ((\texttt{fixpt}\ x) = (x\ (\texttt{fixpt}\ x))) \\ \forall x \forall y\, ((\texttt{cond}\ \texttt{truth}\ x\ y) = x) \\ \forall x \forall y\, ((\texttt{cond}\ \texttt{false}\ x\ y) = y) \end{array}$$

Simple recursive schemes can now be encoded directly. For example, consider introducing the following equation:

```
fact = (fixpt Fact\N\M\ (cond (0 = N) M (Fact (N - 1) (N * M))))
```

which defines a tail recursive version of the factorial program. The following is a sequence of equations which compute the factorial of 3.

```
(fact 3 1) = (fixpt Fact\N\M\ (cond (0 = N) M (Fact (N - 1) (N * M))) 3 1)
           = ((Fact\N\M\ (cond (0 = N) M (Fact (N - 1) (N * M)))) fact 3 1)
           = (cond (0 = 3) 1 (fact (3 - 1) (3 * 1)))
           = (fact 2 3)
            ⋮
           = 6
```

In performing this computation, equations involving λ-conversion and those involving the newly introduced constants have been used. This is an abstract way to view the working of many kinds of functional interpreters.

To illustrate the use of higher-order unification in writing logic programs which transform other programs, consider writing a program which takes as input a functional program whose one argument is always assumed to be a pair of inputs and construct from it the "curried" form of that function. For example, we would like to be able to transform the following program into the factorial program given earlier.

```
(fixpt Fact\P\ (cond (and (pairp P) (0 = (first P)))
                     (second P)
```

```
(cond (pairp P)
      (Fact (pair ((first P) - 1)
                  ((first P) * (second P))))
      error)))
```

If one considers writing this transformation in a language such as Lisp or (first-order) Prolog, a recursive program would need to be written which would descend through the structure of the program, making sure that all occurrences of the bound variable `P` were within expressions of the form `(pairp P)`, `(first P)`, or `(second P)`. Of course, care would need to be exercised in the cases where this descent passed through a part of the program where `P` was bound locally. Such concern for the occurrences of bound variables is an unfortunate aspect of writing such programs in these languages, since bound variable names are merely linguistic conveniences and do not contribute to an understanding of a program's meaning.

The availability of higher-order unification in λProlog, however, permits a very different style of programming where the names and occurrences of particular bound variables is not important. Consider, for example, the following atomic definite clause which defines the λProlog function which relates such curried and uncurried programs.

```
curry (fixpt (Q1\X\ (A (first X) (second X) (pairp X)
                       (R1\R2\ (Q1 (pair R1 R2))))))
      (fixpt (Q2\Y1\Y2\ (A Y1 Y2 truth Q2))).
```

The `curry` predicate relates two terms: the first represents a function of one argument which represents a pair (here, the variable `X`) while the second represents the corresponding function of two variables (here, `Y1` and `Y2`). The higher-order variable `A` represents the body of the first function from which the expressions `(first X)`, `(second X)`, `(pairp X)`, and `(R1\R2\ (Q1 (pair R1 R2)))` have been extracted. This variable is then used to reconstruct the second function by replacing the expresions which have been extracted with `Y1`, `Y2`, `truth`, and `Q2`, respectively. For example, if the first argument of `curry` was instantiated with the factorial program which manipulates pairs of integers, the λProlog interpreter would have determined the value of `A` to be

```
(Z1\Z2\C\Q\ (cond (and C (0 = Z1)) Z2
                  (cond C (Q (Z1 - 1) (Z1 * Z2)))
                        error))).
```

λ-conversion would then have resulted in the second argument of `curry` being bound to

```
(fixpt Q2\Y1\Y2\ (cond (and truth (0 = Y1)) Y2
                       (cond truth (Q2 (Y1 - 1) (Y1 * Y2))
                             error))).
```

Clearly this program could be simplified further by using additional properties of the constant `truth` and such simplification routines could be written easily.

It is important to note that the higher-order unification algorithm is employed here to do the recursive descent through the program structure and that within the unification algorithm it is necessary to be concerned with the bound variables and their occurrences. However, once unification is implemented correctly, the programs written on top of it no longer need to be concerned with such issues explicitly.

5. Using and removing patterns from bindings

As another and rather different illustration of how higher-order unification can be used to analyze programming constructs, consider the follow extension to the functional programming setting of the last section. In defining how functions behave on different arguments, it is often very useful to use *patterns* (such as is used in the ML programming language). If the incoming arguments of the function match a pattern, then the pattern describes how those arguments should be decomposed and how the resulting pieces are used in further computations. If the match with the pattern is not successful, the arguments are compared with other patterns. Consider the following *binding clauses*:

```
append nil L --> L
append (cons X L) K --> (cons X (append L K)).
```

Here the left represents the pattern to be matched with the function's arguments, and the right describes, in terms of the *pattern variables*, what the output should be if the pattern matches successfully.

We can formalize this into our λ-term representation by adding three new constants. First, let `-->` be the constant which builds a simple binding clause from a pattern and an output expression. Next, noticing that the pattern variables are actually local variables per clause, we separately abstract each such variable from the binding clause and then apply the constant `local` which makes such abstractions into a new binding clause. Hence, these two binding clauses are represented as the following λ-terms:

```
(local L\ (append nil L --> L))
(local X\ (local L\ (local K\ (append (cons X L) K -->
                                    (cons X (append L K))))))
```

To make a complete functional description of this program, we place these binding clauses into a list, apply the constant `usepat` to that list, and abstract from that term the function's name. Hence, the following λ-term represents the `append` program which utilizes this binding mechanism.

```
(fixpt (Append\ (usepat
   (cons (local L\ (Append nil L --> L))
   (cons (local X\ (local L\ (local K\ (Append (cons X L) K -->
                                          (cons X (Append L K))))))
        nil)))))
```

Although equations for describing the new constants `-->`, `local`, and `usepat` could be given, we are more interested here in a procedure which can take a program that uses such patterns and automatically transforms it into a program which makes no explicit use of patterns; *i.e.* we wish to replace the use of patterns with explicit uses of destructors.

It is convenient to first consider the case where there is only one pattern in each binding clause. This is not a problem since we can first work with uncurried forms of the function call and then later transform it to the curried form. Consider the following two binding clauses:

```
append (pair nil L) --> L
append (pair (cons X L) K) --> (cons X (append (pair L K))).
```

This is the code for the uncurried form of `append`. Let us focus first on the second clause. If `append` is called with some single input, say `I`, then we could think of the variables `X`, `L`, and `K` as being functions of `I`. Call those functions `XX`, `LL`, and `KK` respectively. Which functions should these be? Precisely those such that the term

```
        (pair (cons (XX I) (LL I)) (KK I))
```

is equal (modulo the equations in the previous section) to `I` itself. Of course, such functions only exist if `I` itself has a certain structure, that is, if `I` is a pair whose first component is a list. The following two λProlog predicates are capable of taking such a pattern and computing from it these functions and this condition.

```
intersect_regions (W\ truth) R R.
intersect_regions R (W\ truth) R.
intersect_regions R1 R2 (W\ (and (R1 W) (R2 W))).

force_pat (I\ I) (W\ truth).
force_pat (I\ nil) (W\ (null W)).
force_pat (I\ (pair (F (first I)) (G (second I)))) R :-
  force_pat F R1, force_pat G R2,
  intersect_regions (W\ (R1 (first W))) (W\ (R2 (second W))) R3,
  intersect_regions (W\ (pairp W)) R3 R.
force_pat (I\ (cons (F (car I)) (G (cdr I)))) R :-
  force_pat F R1, force_pat G R2,
  intersect_regions (W\ (R1 (car W))) (W\ (R2 (cdr W))) R3,
  intersect_regions (W\ (consp W)) R3 R.
```

The meaning of the `intersect_region` predicate is straightforward: it relates three arguments if the third in the intersection of the first two. Its behavior and use in our transformation is very straightforward.

The `force_pat` predicate, however, is more complex, although its declarative reading is still very simple. The intended meaning of the goal `(force_pat F R)` is that the function

F behaves like an identity function over the region specified by the predicate R. That is, if R is true for a value x, then F applied to x is equal to x. This equality is with respect to the equality theory described in the previous section. Given this reading, however, the clauses for force_pat are very nature. The first one states the obvious fact that the function X\X is always an identity. The second clause states that for all objects for which null is true, the I\ nil is the identity function. Since null is true only for nil, the intended meaning holds. Now assume that F is an identity over the region R1 and G is an identity over the region R2. Then the function (I\ (pair (F (first I)) (G (second I)))) is an identity function over the region of pairs where the first component is in the region R1 and the second component is in the region R2. The declarative reading for the last force_pat clause is similar, except the constructors and destructors for lists are used instead of those for pairs. These last two clauses are essentially direct encodings of the equational formulas in $(*)$ of the preceding section.

If we abstract out of the pattern above for append the variable I, we could use the following query to determine the values for the XX, LL, KK, and R functions:

```
force_pat (I\ (pair (cons (XX I) (LL I)) (KK I))) R.
```

The result of asking this query with respect to the preceding code provides the following bindings for the variables in the query:

```
XX = I\ (car (first I))
LL = I\ (cdr (first I))
KK = I\ (second I)
R  = W\(and (pairp W) (consp (first W)))
```

To confirm that these computations are correct, we can substitution these variable bindings into the first argument of the force_pat query and λ-normalize. The resulting expression is then:

```
W\ (pair (cons (car (first W)) (cdr (first W))) (second W)).
```

If this function is applied to any object in our val domain which satisfies the condition expressed in R, that is, if it is a pair whose first component is a list, then the result is equal to that object.

The next step in this transformation to convert a list of binding clauses to a functional program with out such clauses. To do this we use the following trans_bclauses predicate defined by the following definite clauses.

```
trans_bclauses (Fun\I\ nil) (Q\X\ error).
trans_bclauses (Fun\I\ (cons (local X\ (B Fun I X)) (Rest Fun I))) Fp :-
    trans_bclauses (Fun\I\ (cons (B Fun I (XX Fun I)) (Rest Fun I))) Fp.
trans_bclauses (Fun\I\ (cons ((Fun (F I)) -->  (C I)) (Rest Fun I)))
               (Q\X\ (cond (R X) (C X) (Fp Q X))) :-
```

```
    force_pat F R,
    trans_bclauses Rest Fp.
trans_bclauses (Fun\I\ (cons ((Fun (F I)) --> (C0 I (Fun (C1 I))))
                             (Rest Fun I)))
             (Q\X\ (cond (R X) (C0 X (Q (C1 X))) (Fp Q X))) :-
    force_pat F R,
    trans_bclauses Rest Fp.
```

The first argument of this predicate is meant to take the list of binding clauses with the function's name and input argument, the variables `Fun` and `I` respectively, abstracted out. For example, to remove the binding clauses in the uncurried form of the `append` program this first argument would be given the term

```
(Append\I\
   (cons (local L\ (Append (pair nil L) --> L))
   (cons (local X\ (local L\ (local K\ (Append (pair (cons X L) K) -->
                                              (cons X (Append (pair L K)))))))
        nil)))))
```

Given this code, the `trans_bclauses` predicate processes each member of the list under the abstraction. If that list is empty, then the corresponding program is the one that only returns the `error` value. If the list is non-empty and its first binding clause contains a `local` variable declaration then the bound variable is replaced by a function of the input variable. Once local pattern variables have been removed, either one of the last two clauses would now match with the binding clause. The third clause deals with non-recursive binding clauses, that is, the function name `Fun` does not appear on the right of the `-->`. The last clause deals with recursive binding clauses. In each case, the free variable `F` represents the abstracted calling pattern and the variables `C`, `C0` and `C1` represent abstracted parts of the output computations. Both of these clauses also call `force_pat` to determine how the variables in the pattern should be instantiated and over what region the pattern can be resolved. They also call `trans_bclauses` to compute the program, given by `Fp`, which corresponds to the remaining binding clauses. Given the values of `R`, `C` (or `C0` and `C1`), and `Fp`, the corresponding expression which does not contain binding clauses is constructed as the second argument of the `trans_bclauses` predicate.

In the case of the uncurried `append` program, the constructed functional expression would be

```
(Append\I\ (cond (and (pairp I) (null (car I)))
                 (second I)
                 (cond (and (pairp I) (consp (first I)))
                       (cons (car (first I))
                             (Append (pair (cdr (first I)) (second I))))
                error)))
```

With this, the hard part of the transformation is complete. If the curried form of append is desirable, the curry function described earlier could be employed. To do this automatically and to take care of the proper use of the fixpt and usepat constants, the following three predicates provide a convenient interface to the trans_bclauses code.

```
trans_pat (fixpt (Fun\ (usepat (Fp_pat Fun)))) CallingForm
          (fixpt Fp_sans) :-
    trans_bclauses (Fun\I\ (Fp_pat (CallingForm Fun))) Fp_sans.
trans_mono Fp_pat Fp :- trans_pat Fp_pat (F\F) Fp.
trans_bin  Fp_pat Fp :- trans_pat Fp_pat (F\X\Y\ (F (pair X Y))) Uncurried,
                        curry Uncurried Fp.
```

The trans_pat predicate calls trans_bclauses by first removing the occurrences of fixpt and usepat from the given program (its first argument) and then applies the resulting abstractions to CallingForm. The other two predicates are used to determine this calling form. If the defined program is binary, trans_bin is used to specify that the calling form should be (F\X\Y\ (F (pair X Y)), which will convert the binary clauses to its uncurried form, and then to call curry which puts the program back into its curried form. The predicate trans_mono is used to convert programs which are functions of only one argument. Its operation is similar and, in fact, simpler.

6. Transforming Tail Recursion into Iteration

In this section we consider another example of the use of higher-order unification in analysing the structure of programs and in describing transformations between programs based on such an analysis. The example that we consider here involves removing tail recursion in favour of iteration. The task at hand may best be illustrated by a sample application of the transformation. Consider the following simple functional program which computes the sum of two numbers:

```
(fixpt Sum\N\M\ (cond (0 = N) M
                      (Sum (N - 1) (M + 1))))
```

An execution of this program may involve a recursive call to itself. However such a recursive call, if it occurs, would be the last expression that needs to be evaluated. Consequently the recursion in this program may be replaced by a computationally less expensive iteration. The following program in an Algol-like syntax would, for instance, return the sum in result if done were initialized to *true* and loc1 and loc2 were initialized to the numbers whose sum needs to be computed.

```
while not(done) do
  begin if (loc1 = 0)
        then begin done := true ;
                   result := loc2
             end
```

```
        else begin loc1 := loc1 - 1 ;
                   loc2 := loc2 + 1
             end
    end
```

Given our term representation of functional programs, the tail-recursiveness of the above program can easily be recognized. The following term, for instance, would unify only with a term that represents a tail-recursive program

```
(fixpt Fun\X\Y\ (cond (C X Y) (H X Y) (Fun (F1 X Y) (F2 Y))))
```

The idea here is that the argument of `fixpt` in this term imposes a functional structure on any term that it unifies with. The latter term must be such that the only occurrence of the function variable `Fun` being abstracted must be in the 'second arm' of `cond` and, that too, as the principal functor of that arm. It is clear that terms that have such a structure can only correspond to tail-recursive programs.

Before we can describe a transformation of tail-recursive programs of the above sort into iterative programs, we need first to introduce term representations for iterative constructs. One of the key notions in this context is that of commands, and we reserve the (λProlog) type `cmd` for terms that correspond to objects of this syntactic category. Another notion that is of importance is that of locations in a store: we shall use the type `reg` for terms that represent these objects. We now introduce the constants `while` of type `val -> cmd -> cmd`, `if` of type `val -> cmd -> cmd -> cmd`, `:=` of type `reg -> val -> cmd`, `find` of type `loc -> val` and `&` of type `cmd -> cmd -> cmd`; the purpose of `find` is to represent the 'coercion' of a location in the store to its contents, and the last constant is intended to correspond to the operation of 'sequencing' of commands. We shall also use the constants `result`, `done`, `loc1` and `loc2` that are assumed to be of type `reg`. Finally assuming that `:=` and `&` are defined to be infix operators and that `&` has higher precedence than `:=`, we may now represent the iterative program above by the λProlog term

```
(while (not (find done))
      (if ((find loc1) = 0)
         (done := true & result := (find loc2))
         (loc1 := ((find loc1) - 1) & loc2 := ((find loc2) + 1))))
```

The transformation of the term representation of the recursive version of `sum` into the iterative version may now be described via a process of template matching. Assume that a given term unifies with the template shown above. This term may then be converted using the term

```
(while (not (find done))
      (if (C (find loc1) (find loc2))
         (done := true & result := (H (find loc1) (find loc2)))
```

```
            (loc1 := (F1 (find loc1) (find loc2)) &
               loc2 := (F2 (find loc2)))))
```

Unifying the first "template" with the recursive version of `sum` for instance would yield the bindings:

```
C ⟶ X\Y\(X = 0)
H ⟶ X\Y\ Y
F1 ⟶ X\Y\(X - 1)
F2 ⟶ Y\(Y - 1)
```

Substituting these in the second template would yield the term structure corresponding to the iterative version.

The recognition of tail recursion outlined above, and the corresponding conversion to an iterative version, illustrates a novel use of higher-order unification. This kind of use of higher-order unification has been recognised previously by Huet and Lang [7] and has been used there to formalize some of the approaches to program transformations studied by Burstall and Darlington [2]. The notion of template matching that is the basis of this approach is limited in its applicability since only restricted kinds of patterns can be recognized by using it. Consider for instance the following program that computes the greatest common denominator of two numbers:

```
(fixpt Gcd\X\Y\ (cond (1 = X) 1
                      (cond (X = Y) X
                            (cond (X < Y) (Gcd Y X)
                                  (Gcd (X - Y) Y)))))
```

This program is obviously tail-recursive. However the term representation of this program clearly does not unify with the pattern that was used to recognize the tail-recursiveness of `sum`. What is worse is that there is no (second-order) term all of whose instances are term representations of tail-recursive programs and the set of whose instances also contains the term representations of this program and of the sum program.

There is, however, a recursive specification of a class of terms that can be used to recognise the tail-recursiveness of both the programs above. Consider a term of the form `(fixpt Prog)`. In the trivial case this term represents a tail recursive program if `Prog` is of the form `F\X\Y\ (H X Y)` or of the form `F\X\Y\ (F (H X Y) (G X Y))` — i.e. it corresponds to a recursive program in which there are either no recursive calls or there is only a recursive call with modified arguments. However, the term also represents a tail recursive program if `Prog` has the functional structure `F\X\Y\ (cond (C X Y) (H1 F X Y) (H2 F X Y))` where `(fixpt H1)` and `(fixpt H2)` themselves represent tail-recursive programs.

It should now be clear how definite clauses may be combined together with λ-terms to provide a concise specification of the class of terms described above. Such a specification could then be used to recognize the fact that both the sum program and the gcd program are tail-recursive. What is interesting is that the same description also provides us with a means for specifying a transformation into an iterative version of the program. The idea here is as follows. The term (fixpt Prog) transforms into the term (while (not (find done)) Body) where Body is obtained by a transformation of Prog; the intuitive picture is that an iteration (in which done is used to flag the end of the iteration) replaces recursion and the "code" to be iterated over is obtained from the form of the functional program. Now the form of Body can easily be guessed from the cases corresponding to the form of Prog. The case where Prog is of the form F\X\Y\ (H X Y) corresponds to the "bottoming out" of the recursion and hence to iterative code for terminating the iteration. The case when Prog is of the form F\X\Y\ (F (H X Y) (G X Y)) corresponds to resetting of variables within an iteration. Finally the case when Prog has the form of a conditional yields a conditional in iteration.

Putting together these observations and a recognition of some "special" cases in which the iterative code may be simplified, we obtain the following set of definite clauses that specify a richer class of tail-recursion transformation that can actually be specified by the mere use of templates.

```
trans_tailrec (fixpt Prog) (while (not (find done)) G) :-
        trans_body Prog G.
trans_body (F\X\Y\ (H X Y))
           (result := (H (find loc1) (find loc2)) & not_done := false).
trans_body (F\X\Y\ (F (G X Y) Y)) (loc1 := (G (find loc1) (find loc2))).
trans_body (F\X\Y\ (F (G X Y) (H Y)))
           (loc1 := (G (find loc1) (find loc2)) &
            loc2 := (H (find loc2))).
trans_body (F\X\Y\ (F (G X Y) (H X Y)))
           (temp := (find loc1) &
            loc1 := (G (find temp) (find loc2)) &
            loc2 := (H (find temp) (find loc2))).
trans_body (F\X\Y\ (cond (C X Y) (H1 F X Y) (H2 F X Y)))
           (if (C (find loc1) (find loc2)) G1 G2) :-
       trans_body H1 G1, trans_body H2 G2.
```

The predicate trans_tail may be used to transform both the sum program and the gcd program into their respective iterative versions. In attempting to perform either of these tasks the λProlog interpreter would make a non-trivial use of higher-order unification in conjunction with the backchaining mechanism of definite clauses, as the reader can verify.

7. Conclusion

There are several objects whose representation in a manner closely corresponding to their meaning requires the use of a higher-order term language. Some examples of such objects are programs and logical formulas. The ability to reason about such objects is of particular importance in program transformations systems and proof systems. Implementing such systems clearly requires a programming environment in which it is possible to represent higher-order objects in a natural manner, and then to perform manipulations on such representations. The latter task often requires a programming paradigm that is based on search. It is our opinion that there is currently no good computational formalism that supports these requirements. To take one example, the language ML [5] has been used extensively in implementing proof systems. To implement aspects of search, however, the basic computational machinery of this language has to be augmented with an exception handling mechanism. The use of this mechanism often appears to be in conflict with the typing notions that the language provides and so seems to detract from the clarity of the implementation. In contrast, the notion of computation in logic programming languages is based in a fundamental way on the notion of search. In addition, these languages provide another feature that is of importance in the above mentioned implementation tasks, namely the ability to examine the intension, or the manner of description, of objects through unification. The use of logic programming languages in implementing formula or program manipulating systems has, however, been limited; these languages have traditionally been based on first-order logics, and therefore do not possess the mechanisms necessary for representing higher-order objects in a natural manner. By using a higher-order logic to describe a logic programming language, we believe that we have been able to combine the computational machinery of logic programming with a richer representation language and that this has opened up several new and extremely promising applications of the paradigm of logic programming. The purpose of this paper has largely been to provide a number of illustrations to bolster this claim.

8. References

[1] W. F. Clocksin and C. S. Mellish, Programming in Prolog, Springer Verlag, 1981.

[2] J. Darlington and R. Burstall, "A System which Automatically Improves Programs," Proceedings of the Third IJCAI, Stanford, 1973, 479 – 484.

[3] D. M. Gabbay and U. Reyle, "N-Prolog: An Extension to Prolog with Hypothetical Implications. I," Journal of Logic Programming 1, 1984, 319 – 355.

[4] G. Gentzen, "Investigations into Logical Deductions," in *The Collected Papers of Gerhard Gentzen*, edited by M. E. Szabo, North-Holland Publishing Co., 1969, 68 – 131.

[5] M. J. Gordon, A. J. Milner, and C. P. Wadsworth, *Edinburgh LCF*, Springer Verlag, 1979.

[6] G. P. Huet, "A Unification Algorithm for Typed λ-Calculus," Theoretical Computer

Science 1, 1975, 27 – 57.

[7] G. P. Huet and B. Lang, "Proving and Applying Program Transformations Expressed with Second-Order Patterns," Acta Informatica 11, (1978), 31 – 55.

[8] D. Miller, "A Theory of Modules for Logic Programming," Proceedings of the Symposium on Logic Programming, 1986, 106 – 115.

[9] D. Miller and G. Nadathur, "Some Uses of Higher-Order Logic in Computational Linguistics," Proceedings of the 24th Annual Meeting of the Association for Computational Linguistics, 1986, 247 – 255.

[10] G. Nadathur, "A Higher-Order Logic as the Basis for Logic Programming," Ph.D. Dissertation, University of Pennsylvania, May 1987.

[11] L. C. Paulson, "Natural Deduction as Higher-Order Resolution," The Journal of Logic Programming 3(3), 1986, 237 – 258.

[12] D. H. D. Warren, "Higher-order extension to PROLOG: are they needed?," Machine Intelligence 10, 1982, 441 – 454.