HIGHER-ORDER LOGIC PROGRAMMING

Dale A. Miller and Gopalan Nadathur
Computer and Information Science
University of Pennsylvania
Philadelphia PA 19104–6389 USA

Abstract: In this paper we consider the problem of extending Prolog to include predicate and function variables and typed λ-terms. For this purpose, we use a higher-order logic to describe a generalization to first-order Horn clauses. We show that this extension possesses certain desirable computational properties. Specifically, we show that the familiar operational and least fixpoint semantics can be given to these clauses. A language, λProlog that is based on this generalization is then presented, and several examples of its use are provided. We also discuss an interpreter for this language in which new sources of branching and backtracking must be accommodated. An experimental interpreter has been constructed for the language, and all the examples in this paper have been tested using it.

Section 1: Introduction

The introduction of higher-order objects has been a major consideration in the realm of functional programming, and indeed these have proved to be very valuable in languages such as Lisp, Scheme, and ML. It is of interest therefore to consider the possibility of introducing such objects into a logic programming language. We examine this issue in this paper.

It is our belief that any attempt at providing a logic programming language like Prolog with the ability to deal with higher-order objects must be based on an extension to the underlying logic. Consider for example the facility Lisp provides for constructing lambda expressions which can be passed as parameters and can, later, be used as programs. In the setting of logic programming this corresponds to permitting predicate variables which may be instantiated by lambda expressions and allowing goals to be expressions that need to be lambda normalized before they are invoked. Given its logical basis, this feature is not directly available in Prolog. However, an argument may be made (eg. [D. H. Warren, 1982]) that no extension to Prolog or to the underlying logic is necessary by demonstrating how certain uses of this feature can be encoded in the first-order language. In our opinion, such an argument is inappropriate. First of all, it is desirable to provide for higher-order features such as the ones above in a natural and theoretically well understood fashion and from this perspective the ability to encode certain uses of predicate variables in the existing language is clearly not sufficient. Furthermore, the nature of objects in the paradigm of logic programming is somewhat different from that in the paradigm of functional programming. The question of what it means to have genuine higher-order objects in a logic programming language, therefore, is itself open to examination, and it seems that a study of this question should rely on an underlying logic.

In this paper we present a logic programming language that permits functions and predicates as objects. This language is based on a logic that uses the mechanism of the typed λ-calculus for

This work has been supported by NSF grants MCS8219196-CER, MCS-82-07294, AI Center grants NSF-MCS-83-05221, US Army Research office grant ARO-DAA29-84-9-0027, and DARPA N000-14-85-K-0018.

constructing predicate and function terms and permits a quantification over such constructions. Using this logic we find that we are able to describe a higher-order generalization of the first-order Horn clauses which shares many computational properties with its first-order counterpart. These clauses can be used to define a programming language that allows function and predicate variables and whose term structure is now that of λ-terms. One consequence of this is the provision of Lisp-like features. However, extending the notion of terms also gives the language a much richer set of data structures, and the operations of λ-conversion and unification on these provides a computational paradigm not found earlier in either logic or functional programming paradigms. It must be pointed out that the features that are provided are higher-order in a strictly logical sense. They do not include features popularized by, for example, the `setof` and `bagof` constructs [D. H. Warren, 1982]; these extensions are perhaps better classified as meta or control level extensions since they involve endowing a logic programming language with an understanding of its own ability to prove. We do not focus on these meta level aspects in this paper, but we note that they may be added to our language in a manner analogous to their addition to Prolog.

The structure of this paper is as follows. In Section 2 we describe the higher-order logic that we use as the basis of our language. Following this, in Section 3, we present our generalization to Horn clauses and discuss their formal properties. We have designed a programming language which includes not only these higher-order characteristics but also features like parametric polymorphic types and modules, that have already been found useful in other contexts (eg. ML and [Mycroft and O'Keefe, 1985]). This language, called λProlog, is described in Section 4, where several examples of its use are also presented. Finally, Section 5 discusses theorem-proving in the context of our clauses, and then uses this to describe an interpreter for λProlog. An experimental interpreter has been built along these lines, and all the examples in Section 4 and [Miller and Nadathur, 1985] have been tested on it.

Section 2: A Higher-Order Logic

The term "higher-order logic," as it is often understood, pertains to a logic whose language admits function and predicate variables, and in which such variables are interpreted as ranging over arbitrary functions and relations on any given domain. By virtue of Gödel's incompleteness theorems, it is known that a logic of this kind is not recursively axiomatizable and that its set of valid sentences is not effectively enumerable. Such a logic is not very interesting from our viewpoint, since our purpose is to use theorem-proving as the method of computation. Fortunately there is a higher-order logic that involves a weaker notion of quantification that can be recursively axiomatized. The Simple Theory of Types, presented by Church in [Church, 1940], is a typed λ-calculus formulation of this logic. The higher-order logic, called $\mathcal{T}$, that we use as the basis of our programming language is derived from the Simple Theory of Types. In this section we present a brief exposition of $\mathcal{T}$. A detailed account of the logic and its proof-theoretic properties are beyond the scope of this paper, and the interested reader is referred to [Church, 1940] and [Miller, 1983].

The language of $\mathcal{T}$ is a *typed* language in the sense that each well formed formula of the system has associated with it a type symbol. We assume that we are given a set $\mathcal{S}$ of *sorts* or *primitive types*, a set $\mathcal{V}$ of *type variables*, and a set $\mathcal{C}$ of *type constructors* where each type constructor has a unique positive arity. The types of $\mathcal{T}$ are then defined inductively by the following rules:

(1) Each sort and each type variable is a type.
(2) If c is an n-ary type constructor and $t_1, \ldots, t_n$ are types, then $(c\ t_1 \ldots t_n)$ is a type.
(3) If t_1 and t_2 are types, then $t_1 \rightarrow t_2$ is a type.

The set $\mathcal{S}$ must contain the sorts o and i; o is intended to be the type of propositions and i is the type of individuals. These are the only sorts that are necessary in the logic, but we shall assume here that $\mathcal{S}$ also contains the sort *int* for integer. $\mathcal{V}$ must be a denumerable set and we assume that α and β are included amongst its members. We also assume that $\mathcal{C}$ contains the type constructor *list* of arity 1. A type in which type variables occur is intended to correspond to the set of all its type instances that do not contain any type variables; a type t' is said to be a *type instance* of another type t just in case it is obtained from t by replacing simultaneously some of the variables in t with types. The type $t_1 \rightarrow t_2$ is also called a *function* type. We adopt the convention that $\rightarrow$ is right associative, *i.e.* we read $t_1 \rightarrow t_2 \rightarrow t_3$ as $t_1 \rightarrow (t_2 \rightarrow t_3)$. A type $t_1 \rightarrow t_2$ in which no type variables occur is intended to be the type of functions whose domain is of type t_1 and whose codomain is of type t_2.

We now turn to the well formed formulas of $\mathcal{T}$. Here we assume that we are given a set of constants and a denumerable set of variables, and that each element of these sets is specified with a particular type. The set of constants contains at least the following symbols that are referred to as the *logical constants* of $\mathcal{T}$:

Constant	Type
$\wedge$	$o \rightarrow o \rightarrow o$
$\vee$	$o \rightarrow o \rightarrow o$
$\supset$	$o \rightarrow o \rightarrow o$
$\sim$	$o \rightarrow o$
true	o
Π	$(\alpha \rightarrow o) \rightarrow o$
Σ	$(\alpha \rightarrow o) \rightarrow o$

The remaining constants are called the *nonlogical constants*. The following is a familiar set of such constants that we shall find occasion to use in this paper:

Constant	Type
cons	$\alpha \rightarrow (\mathit{list}\ \alpha) \rightarrow (\mathit{list}\ \alpha)$
nil	$(\mathit{list}\ \alpha)$
$+$	$\mathit{int} \rightarrow \mathit{int} \rightarrow \mathit{int}$
$-$	$\mathit{int} \rightarrow \mathit{int} \rightarrow \mathit{int}$
$*$	$\mathit{int} \rightarrow \mathit{int} \rightarrow \mathit{int}$

The formulas of $\mathcal{T}$, with their respective types, are defined inductively by the following rules:
(1) A variable of type t is a formula of type t.
(2) A constant whose specified type is t is a formula of type t', for any t' which is a type instance of t. Thus *cons* is a formula of type $\mathit{int} \rightarrow (\mathit{list}\ \mathit{int}) \rightarrow (\mathit{list}\ \mathit{int})$ as well as of type $(\mathit{list}\ \beta) \rightarrow (\mathit{list}\ (\mathit{list}\ \beta)) \rightarrow (\mathit{list}\ (\mathit{list}\ \beta))$.
(3) If f_1 is a formula of type $t_1 \rightarrow t_2$ and f_2 is a formula of type t_1, then the *application* of f_1 to f_2, written $(f_1\ f_2)$ is a formula of type t_2. We assume that application is left

associative, *i.e.* we read $(f_1\ f_2\ f_3)$ as $((f_1\ f_2)\ f_3)$. Functions of many arguments are represented here in a curried form.

(4) If x is a variable of type t_1 and f is a formula of type t_2 then the *abstraction* of f by x, written $(\lambda x\ f)$, is a formula of type $t_1 \rightarrow t_2$; the abstraction is said to *bind* x and its *scope* is said to be f.

A formula in which no variables occur free is said to be a *closed* formula; an occurrence of a variable, x, in a formula is a *free* occurrence if it is not in the scope of an abstraction that binds x. A type symbol is considered to occur in a formula if it occurs in the type of some variable or constant of the formula. If a formula is the result of substituting types for some of the type variables in another formula, then the former is said to be a type instance of the latter. A formula in which type variables occur is to be interpreted as a scheme - it represents the set of all its type instances in which no type variables occur. Type variables thus provide a form of quantification over types. However, no explicit quantification is provided for, and the implicit universal quantification of a type variable that occurs in a formula is obviously over the whole formula. A stronger type system and a better formalization of the formulas that we have presented here is perhaps obtained through the use of explicit type quantification as in the second-order lambda calculus ([Reynolds, 1985], [Fortune, Leivant and O'Donnell, 1983]), but we do not pursue this aspect in this paper. The use that we make of type variables does not add anything to the logic, but it does provide a valuable form of polymorphism in the programming language that we shall define. In that context type constructors conspire with type variables to provide a form of *parametric polymorphism*. For instance, *cons* may be used to construct many different kinds of lists, the elements of each such list being homogenous.

λ-conversion plays an important role in $\mathcal{T}$. Let x be a variable and let t and A be terms. If there is no abstraction in A in whose scope x appears free and which also binds a free variable of t then we say that t is *free for x in A*. We write $A[t/x]$ to represent the result of replacing all free occurrences of x in A by t; obviously this is a meaningful operation only if t and x have the same type and t is free for x in A. The following three operations now comprise λ-conversion.
α-conversion: Replacing $(\lambda x\ A)$ with $(\lambda y\ A[y/x])$ provide y is free for x in A.
β-conversion: Replacing $(\lambda x\ A)t$ with $A[t/x]$ and vice versa provided t is free for x in A.
η-conversion: Replacing A with $\lambda z(Az)$ and vice versa if A has type $\alpha \rightarrow \beta$ and z has type α, provided z is not free in A.

A formula A is said to be convertible to another formula B if B can be obtained from A by a sequence of λ-conversions. Two formulas are considered equal if they are each convertible to the other; further distinctions can be made between formulas in this sense by omitting the rule for η-conversion, but we feel that these are not important in our context. We shall say here that a formula is a *λ-normal formula* if it has the form

$$\lambda x_1 \ldots \lambda x_n\ (h\ t_1\ \ldots\ t_m) \qquad \text{where } n, m \geq 0,$$

where h is a constant or variable, $(h\ t_1\ \ldots\ t_m)$ does not have a function type, and, for $1 \leq i \leq m$, t_i also has the same form. We call the list of variables $x_1, \ldots, x_n$ the *binder*, h the *head* and the formulas $t_1, \ldots, t_m$ the arguments of such a formula. It is known that every formula, A, can be converted to a λ-normal formula that is unique upto α-conversions. We call such a formula a λ-normal form of A and we use $\lambda norm(A)$ to denote any of these alphabetic variants.

The type o plays a special role in $\mathcal{T}$. A formula with a function type of the form $t_1 \rightarrow \ldots \rightarrow t_n \rightarrow o$ is also classified as a *predicate* of n arguments whose i^{th} argument must be of type t_i. Predicates are use to denote sets and relations. For example, predicates of type $int \rightarrow o$ represent sets of integers, predicates of type $(int \rightarrow o) \rightarrow o$ represent sets of sets of integers, and predicates of the type $\alpha \rightarrow (list\ \alpha) \rightarrow o$ represent binary relations between objects of any type a in which no type variables occur and the corresponding type $(list\ a)$. Formulas of type o are called *propositions*; notice that these formulas must have an empty binder. The logical constants $\wedge$, $\vee$, and $\supset$ correspond to the familiar propositional connectives, and we shall adopt the customary infix notation for these. The symbols Π and Σ are used in conjunction with the abstraction operation to represent universal and existential quantification over propositions: $\forall x\ f$ is an abbreviation for $\Pi(\lambda x\ f)$ and $\exists x\ f$ is an abbreviation for $\Sigma(\lambda x\ f)$. Derivability in $\mathcal{T}$, denoted by $\vdash_{\mathcal{T}}$, is a notion that pertains to propositions and is an extension of the notion for first-order logic. The axioms of $\mathcal{T}$ are the substitution instances of the propositional tautologies, the formula $\forall x\ Bx \supset Bt$, and the formula $\forall x\ (Px \wedge Q) \supset \forall x\ Px \wedge Q$. The rules of inference of the system are *Modus Ponens*, *Universal Generalization*, *Substitution*, and *λ-conversion*. λ-conversion is essentially the only rule in $\mathcal{T}$ that is not in first-order logic, but combined with the richer syntax of formulas in $\mathcal{T}$ it makes more complex inferences possible. $\mathcal{T}$, unlike the Simple Theory of Types, is a logic that is not *extensional*; *i.e.* given two 1-ary predicates P and Q it may be possible to prove $\forall x\ (Px \equiv Qx)$ in $\mathcal{T}$ without being able to prove that P and Q are equal.

We are interested in $\mathcal{T}$ because it possesses several properties that make it a suitable basis for the kind of programming language that we desire. It provides a mechanism for constructing function and predicate terms and for permitting variables to range over such constructions, and this was our main reason for looking for a higher-order logic. Further the proof-theory for $\mathcal{T}$ bears a close resemblance to that of first-order logic; for instance there is a generalization to Herbrand's Theorem [Miller, 1983] that holds for $\mathcal{T}$. This property shall be of importance when we consider the task of designing an interpreter for our language. Finally there is a sublogic of $\mathcal{T}$ that generalizes the definite clauses of first-order logic while preserving several of their computational properties. It is this sublogic that we examine in the next section, and that we use later to define our programming language.

Section 3: Higher-Order Definite Clauses and their Properties

We shall henceforth assume that we have a fixed set K of nonlogical constants. The *positive Herbrand Universe* is identified in this context to be the set of all the λ-normal formulas that can be constructed using the nonlogical constants in K and no logical constants other than *true*, $\wedge$, $\vee$ and Σ. We use the symbol $\mathcal{H}^+$ to denote this set. Propositions in this set are of special interest to us. We shall use, perhaps with subscripts, the symbol G to denote an arbitrary such proposition throughout this paper. Notice that the head of such a formula is either a predicate constant or variable or one of the constants *true*, $\wedge$, $\vee$, and Σ. Of these formulas we single out those that have nonlogical constants as their heads. We shall call such formulas *atoms*, and we use the symbol A uniformly to denote an atom.

A *(higher-order) definite clause* is defined to be the universal closure of a formula of the form $G \supset A$, *i.e.* the formula $\forall \bar{x}\ (G \supset A)$ where $\bar{x}$ is an arbitrary listing of all the free variables in G and A. These clauses are our generalization of the Horn clauses of first-order logic. There are certain relationships between these that should be pointed out. First-order Horn clauses are contained

in our definite clauses under an implicit encoding. This encoding essentially assigns types to the first-order terms and predicates: variables and constants (*i.e.* 0-ary function symbols) are assigned the type i, function symbols of arity $n > 0$ are assigned the type $i \to \ldots \to i \to i$, with $n+1$ occurrences of i, and n-ary predicate symbols are assigned the type $i \to \ldots \to i \to o$, with n occurrences of i. Looked at differently, our definite clauses contain within them a polymorphic many-sorted version of first-order Horn clauses. The formula on the left of the $\supset$ in a higher-order definite clause may contain nested disjunctions and existential connectives. This generalization may be dispensed with in the first-order case because of the existence of appropriate normal forms. For the higher-order case, it is more natural to retain the embedded disjunctions and existential quantifications since substitutions have the potential for reintroducing them. Finally λ-terms may occur in the higher-order clauses and the quantifications in these clauses may involve function and predicate variables. This is a genuine extension provided by our clauses, and is the very reason why we study them.

Parallel to the first-order case, we wish to accord a computational interpretation to our definite clauses. Let $\mathcal{P}$ be a set of definite clauses, and let G have no type variables in it. We want to think of $\mathcal{P}$ as a program and of G as *query* or a *goal*. The computation involved is then to be that of answering the query. The sense in which the query is to be answered may be made precise as follows. Let us define a substitution to be a finite sequence of pairs, $\varphi = \langle\langle x_1, t_1\rangle, \ldots, \langle x_n, t_n\rangle\rangle$, where the x_i's are distinct variables, and, for each i, t_i is a formula of the same type as x_i; φ is said to be a substitution for $x_1, \ldots, x_n$ and its *range* is the set $\{t_1, \ldots, t_n\}$. The application of φ to a formula B, written as φB, is defined to be $\lambda norm([\lambda x_1 \ldots \lambda x_n\ B]t_1 \ldots t_n)$. Let $\bar{y}$ be an arbitrary listing of all the variables free in G. Now, we want the query G to be answered affirmatively if $\mathcal{P} \vdash_\mathcal{T} \exists \bar{y}\ G$ and we also want an affirmative answer to be accompanied by a substitution for $\bar{y}$ such that $\mathcal{P} \vdash_\mathcal{T} \varphi G$.

The latter may not always be possible if $\mathcal{P}$ is any arbitrary set of formulas. However, the following theorem assures us that it is indeed possible for a collection of definite clauses. Here and in the rest of the paper we reserve the terms *positive* substitution for one whose range is a subset of $\mathcal{H}^+$, and *closed* substitution for one whose range consists of closed formulas. We also use the symbol $\mathcal{P}$ uniformly to denote a (possibly empty) set of definite clauses, and we write $[\mathcal{P}]$ to denote the set of formulas of the form $\varphi(G \supset A)$ where $\forall \bar{x}\ (G \supset A)$ is a type instance of a formula of $\mathcal{P}$ which contains no free type variables, and φ is a positive, closed substitution for $\bar{x}$.

Theorem 1: Let $G \in \mathcal{H}^+$ be a closed proposition that has no type variables in it. Then the following are true:

(1) If G is $G_1 \wedge G_2$ then $\mathcal{P} \vdash_\mathcal{T} G$ if and only if $\mathcal{P} \vdash_\mathcal{T} G_1$ and $\mathcal{P} \vdash_\mathcal{T} G_2$.

(2) If G is $G_1 \vee G_2$ then $\mathcal{P} \vdash_\mathcal{T} G$ if and only if $\mathcal{P} \vdash_\mathcal{T} G_1$ or $\mathcal{P} \vdash_\mathcal{T} G_2$.

(3) If G is $\Sigma\ B$ then $\mathcal{P} \vdash_\mathcal{T} G$ if and only if there is a closed formula $t \in \mathcal{H}^+$ such that $\mathcal{P} \vdash_\mathcal{T} \lambda norm(Bt)$.

(4) If G is an atom then $\mathcal{P} \vdash_\mathcal{T} G$ if and only if there is a formula $G_1 \supset G \in [\mathcal{P}]$ such that $\mathcal{P} \vdash_\mathcal{T} G_1$.

The proof of this and the other theorems in this paper may be obtained from the results in [Miller and Nadathur, 1986]. As a consequence of this theorem we may attribute a procedural interpretation to a clause. Consider the definite clause $\forall \bar{x}(G \supset A)$. G may either be *true* or a compound formula containing conjunctions, disjunctions, and existential quantifiers. If G is *true*, then the clause is logically equivalent to $\forall \bar{x} A$, and is to be interpreted as a fact. Otherwise

we interpret it as a procedure declaration, where the non-logical head of A is the name of the procedure being defined, and G is the procedure body which is to be used to compute it. Note that by virtue of this theorem we need only consider positive substitutions in order to establish a goal from a set of definite clauses. This fact, in conjunction with the observation that a positive substitution when applied to an element in $\mathcal{H}^+$ produces another element in $\mathcal{H}^+$, enables us to define, even in the presence of predicate variables, a theorem-prover for this sublogic that is based on this procedural interpretation of clauses. We shall consider such a theorem-prover shortly.

It is possible to explicate the meaning of a set of definite clauses in a more direct manner by associating with it a set of atoms. The idea is similar to that used in the first-order case (see [Apt and van Emden, 1982] and [van Emden and Kowalski, 1976]) and may be made precise in the following manner. Let us define an *interpretation* to be any set of closed atoms in which no type variables occur. Relative to an interpretation I we may define a *derivation sequence* to be a finite sequence $G_0, G_1, \ldots, G_n$ of closed propositions in $\mathcal{H}^+$ in which no type variables occur and for each $i \leq n$,

(1) G_i is *true*, or

(2) G_i is an atom and G_i λ-converts to some member of I, or

(3) G_i is $G_i^1 \vee G_i^2$ and there is a $j < i$ such that G_j is G_i^1 or G_j is G_i^2, or

(4) G_i is $G_i^1 \wedge G_i^2$ and there are $j, k < i$ such that G_j is G_i^1 and G_k is G_i^2, or

(5) G_i is ΣG and there is a closed formula $t \in \mathcal{H}^+$ and a $j < i$ such that G_j is $\lambda norm(Gt)$.

If G is the last element of such a sequence, we say that I *satisfies* G and we denote this relation by $I \models G$.

Given a set of definite clauses $\mathcal{P}$, we associate with it a mapping $T_{\mathcal{P}}$ from interpretations to interpretations which is such that $A \in T_{\mathcal{P}}(I)$ if and only if there is a formula $G \supset A \in [\mathcal{P}]$ such that $I \models G$. It is not difficult to see that $T_{\mathcal{P}}$ is monotone and continuous on the set of all interpretations. $T_{\mathcal{P}}$ therefore has a least fixed point which is given by $T_{\mathcal{P}}^{\infty}(\emptyset) = \bigcup_{n=0}^{\infty} T_{\mathcal{P}}^{n}(\emptyset)$. It is this subset of $\mathcal{H}^+$ that we think of as being determined by $\mathcal{P}$, and we call it the *denotation* of $\mathcal{P}$. The computation that is involved in answering a query G may be viewed as that of determining whether there is a closed substitution instance of G that is satisfied in the denotation of $\mathcal{P}$. The consistency of this view with the earlier operational view is the content of the following theorem:

Theorem 2: Let G be a closed formula with no type variables. Then $T_{\mathcal{P}}^{\infty}(\emptyset) \models G$ if and only if $\mathcal{P} \vdash_{\mathcal{T}} G$.

Section 4: The λProlog language

Our programming language, λProlog, is based on higher-order definite clauses. Since their underlying logics are similar, we find it convenient to adopt several features of the syntax of Prolog in λProlog. Symbols that begin with capital letters, both in clause definitions and in type definitions, are treated as variables. All other symbols represent constants. The symbols `,`, `;`, and `:-` are used for $\wedge$, $\vee$ and $\supset$ respectively, and clauses are written backwards. Variables occuring in clauses are assumed to be implicitly universally quantified.

There are, however, a few differences. We need to represent λ-terms and the symbol `\` is reserved for this purpose: $\lambda X\ A$ is written in λProlog as `X \ A`. The constant `sigma` is reserved for Σ. A curried notation is adopted since it is especially convenient in our context, and application is represented by juxtaposition. Types must be associated with every (term) constant and variable and this is achieved via a type declaration that has the format `type token logical_type`.

We have found it useful to organise declarations into modules and have introduced this notion as a structuring concept in λProlog. Modules are, in our context, named environments within which operator and type declarations may be associated with tokens, and defining clauses may be presented for predicate constants. The following is an illustration of this structure:

```
module tiny.
op 255 xfx :-.
op  40 xfx =.
type :- o -> o -> o.
type =  A -> A -> o.
onep X :- X = 1.
identity_fun F :- (X\ X) = F.
```

Operator declarations override the default prefix application precedence, and are similar to those in Prolog: `op 225 xfx :-` corresponds to op(225,xfx,:-) in Dec10 Prolog syntax. Type and operator declarations are considered attributes of a module and are not side effects. In general, very little type information needs to be given, since most of it can be inferred from the context. The rules for inferring types are essentially those used in ML [Milner, 1978]. In performing such an inference, we assume that all occurrences of a bound variable within the scope of its abstraction and all occurrences of a constant in a module have the same type. As an instance of such a type determination, the types of the constants `onep` and `identity_fun` can be inferred to be `int -> o` and `(A -> A) -> o`, respectively. Our module parser is able to perform such a type determination, and in this case it assumes that these are also part of the type declarations in the module. The module `tiny` also associates defining clauses with the predicates `onep` and `identity_fun`. This module, thus, defines eight associations: two operator specifications, four type declarations (two explicit, two inferred), and two predicates with their definite clauses.

A module may also import several other modules. The effect of this operation is to make available the operator and type declarations and the definite clauses of the imported modules in the module being defined. The precise logical characterization of this operation with regard to the clauses depends on an assimilation of implication into the body of definite clauses, and an attempt in this direction may be found in [Miller, 1986].

We assume, in the rest of this section, that the module `basics` contains all type and operator declarations for many standard Prolog logical constructions. The following module, which imports `basics`, then provides an illustration in λProlog of some standard list manipulation programs.

```
module lists.
import basics.
type cons A -> (list A) -> (list A).
type nil (list A).
append nil K K.
append (cons X L) K (cons X M) :- append L K M.
memb X (cons X L).
memb X (cons Y L) :- memb X L.
member X (cons X L) :- !.
member X (cons Y L) :- member X L.
```

Here, cut (!) is intended to have the same operational meaning as it does in Prolog, *i.e.* it removes all backtracking points. The following type information is inferred and is also assumed to be a part of this module's definition.

```
type append (list A) -> (list A) -> (list A) -> o.
type memb (list A) -> (list A) -> o.
type member (list A) -> (list A) -> o.
```

One of the novelties of λProlog is the provision of predicate variables. The following module offers an illustration of this facet:

```
module age.
import basics lists.
type age i -> int -> o.
type have_property (A -> o) -> (list A) -> (list A) -> o.
have_property P (cons X L) (cons X K) :- P X, have_property P L K.
have_property P (cons X L) K :- have_property P L K.
have_property P nil nil.
mappred P (cons X L) (cons Y K) :- P X Y, mappred P L K.
mappred P nil nil.
have_age L K :- have_property (Z\(sigma X\(age Z X))) L K.
same_age L K :- have_property (Z\(age Z A)) L K.
age sue  24.
age bob  23.
```

This module defines the predicate **have_property** whose first argument must be a predicate and is such that (**have_property P L K**) is true if K is some sublist of L and all the members in K satisfy the property expressed by the predicate P. Using **have_property** the predicate **have_age** is defined such that (**have_age L K**) is true if K is a sublist of the objects in L which have an age. Notice that there is an explicit quantifier imbedded in the predicate used to define **have_age**. The predicate (**Z\(sigma X\(age Z X))**), which may be written in logic as $\lambda z \exists x\ age(z,x)$, is true of an individual if that individual has an age. The predicate **same_age** whose definition is obtained by dropping that quantifier defines a slightly different property; (**same_age L K**) is true only when the objects in K have, in addition, the same age.

In the cases considered above, predicate variables that appeared as the heads of goals were fully instantiated before the goal was invoked. This kind of use of predicate variables is similar to the use of apply and lambda terms in Lisp; the λ-contraction followed by the goal invocation essentially simulates the apply operation. However, the variable head of a goal need not always be fully instantiated, and in such cases there is a question concerning what substitution should be returned. Consider for example the query (**P bob 23**). One value that may be returned for P is **X\Y\(age X Y)**. But there are many more substitutions which also satisfy this goal; **X\Y\(X = bob, Y = 23)**, **X\Y\(Y = 23)**, **X\Y\(age sue 24)**, etc. are all terms that could also be picked.

Clearly there are too many such substitutions to pick from and then backtrack over. Our decision is to use only the substitution that corresponds to the largest "extension" in such cases; in the above case, for example, the term **X\Y\true** would be picked. It is possible to make such a choice without adding to the incompleteness of an interpreter, and we comment on this issue in Section 5. For the moment we note that this decision does not trivialize the use of predicate

variables. Assume for instance that a predicate concept of type `(i -> o) -> o` has been defined. Then the query `concept P, P t` would still be a meaningful one. This query would entail looking for a predicate term which is a concept, and then asking if `t` is in its extension.

As we noted, the addition of predicate variables is a little like adding Lisp's notions of apply and lambda expressions to Prolog. The additions of function variables and higher-order unification, however, are in an entirely new direction. Consider adding the following definite clauses at the the end of the module `lists`.

```
mapfun F (cons X L) (cons (F X) K) :- mapfun F L K.
mapfun F nil nil.
```

The type for `mapfun` would be inferred to be `(A -> B) -> (list A) -> (list B) -> o`. Given the goal `(mapfun (X\(g X X)) (cons a (cons b nil)) L)`, our interpreter would return the list `(cons (g a a) (cons (g b b) nil))` as the answer substitution for `L`. In other words, if the first two arguments are instantiated then the list that results from applying the first argument to each element of the second would be returned as the value of the third argument. Notice that mapping a function over a list is quite different from mapping a predicate over a list as in the `mappred` procedure defined earlier. In the latter case the idea of applying a predicate, say `P`, to an argument, say `X`, entails creating a new goal – the λ-normal form of `(P X Y)` for some variable `Y`. The value placed in the list is an instance of `Y` that enables this goal to be derived. In mapping a function over a list, no new goals are constructed. The function is simply applied to the argument and the resulting λ-normal form is the value entered into the list. Since mapping a function is weaker than mapping a predicate, the problem of discovering functions which successfully map a list into another list is better defined and does not always permit trivial solutions. For example, consider the goal,

```
(mapfun F (cons a (cons b nil)) (cons (g a a) (cons (g a b) nil))).
```

Here there is exactly one substitution for `F` which satisfies this goal, namely `F` gets `X\(g a X)`. Notice that backtracking may occur on unifying substitutions as well. In searching for an answer substitution a depth-first interpreter would first consider unifying `(F a)` with `(g a a)`. There are four possible substitutions for `F` that are unifiers:

```
X\(g X X)    X\(g a X)    X\(g X a)    X\(g a a).
```

If any of these other than the second is picked, the interpreter would fail in matching `(F b)` with `(g a b)`, and would therefore have to backtrack.

λ-terms obviously provide much richer data structures than those afforded by simple first-order terms, and there are situations in which this richness in λProlog can be exploited. Examples of its use in the realms of knowledge representation and natural language semantics may be found in [Miller and Nadathur, 1985] and [D. S. Warren, 1983]. Another realm in which it is useful is that of program transformations. [Huet and Lang, 1978] indicates how program transformation algorithms may be written rather directly by encoding program structures using λ-terms, and then using higher-order unification. The following module presents a program that may be used to do the *unfolding* transformation.

```
type if (env -> bool) -> A -> A -> A.
type while (env -> bool) -> (env -> env) -> (env -> env).
type unfold (A -> (env -> env)) -> (A -> (env -> env)) -> o.
unfold (X\(while (Cond X) (Prog X)))
       (X\(if (Cond X)
              (E\(while (Cond X) (Prog X) (Prog X E)))
              (F\F))).
```

The predicate **unfold** can be used to expand a **while**-loop into an **if** construction. Consider the goal,

```
unfold (W\(while (lessthan W 10) (advance W 1))) Q.
```

The unique solution to this goal returns the following substitution for Q that is computed entirely within the unification process.

```
W\(if (lessthan W 10) E\(while (lessthan W 10) (advance W 1) (advance W 1 E)) F\F)
```

The clause defining **unfold** is used with the variables Cond and Prog bound to `W\(lessthan W 10)` and `U\(advance U 1)` respectively in this computation.

The provision of polymorphic types *and* function types adds an interesting complexity to the language. Consider the following module.

```
module interpreter.
import basics lists.
interp H true.
interp H (G1, G2) :- interp H G1, interp H G2.
interp H (G1; G2) :- interp H G1; interp H G2.
interp H (sigma G) :- interp H (G X).
interp H A :- memb Clause H, instan Clause (A :- G), interp H G.
instan (pi B) C :- instan (B X) C.
instan C C.
```

Here, **interp** is a two place predicate. If Cs is a list of closed definite clauses and G is a goal then `(interp Cs G)` succeeds if and only if there is a proof of an instance of G from the clauses in Cs. This program constitutes an interpreter for that subset of λProlog in which type variables are not permitted in definite clauses. In the first clause of **instan**, the variable B has type `A -> o` for some type variable A. When this clause is invoked, this type variable must be instantiated. A value for that type variable may only be obtained by examining the term with which it is getting unified. In other words, this is a case where a function type needs to be determined dynamically. When **instan** is called from **interp** there is a fully instantiated term in its second argument, so this does not constitute a problem. It may, however, be the case that when a type variable needs to be determined at runtime the term that needs to be examined is not instantiated in such a way as to provide an actual type. This would happen, for example, if **instan** is invoked with only its second argument instantiated. Such a situation may cause a problem for the interpreter, and we discuss it further in the next section.

Section 5: An Abstract Interpreter for Definite Clauses

We now desire a mechanism for finding proofs in $\mathcal{T}$ for a goal of the form $\exists \bar{x} G$ from a set of definite clauses $\mathcal{P}$. In its abstract description we expect such a mechanism to be complete,

i.e. it should return a positive answer whenever a derivation does exist. Furthermore, whenever it provides a positive answer, it should also provide a substitution φ for $\bar{x}$ such that $P \vdash_{\mathcal{T}} \varphi G$. The structure of such a mechanism is easily obtained from Theorem 1 in Section 3. However we desire to describe it at a sufficient level of detail so that it may form the basis of an interpreter for λProlog. In order to do so we need to consider briefly the problem of unifying typed λ-terms.

Let us call a pair of terms of the same type a *disagreement pair*. A *disagreement set* is a finite set $\{\langle t_1, s_1\rangle, \ldots, \langle t_n, s_n\rangle\}$ of disagreement pairs, and a *unifier* for this disagreement set is a substitution θ such that, for each $i \leq n$, θt_i is λ-convertible to θs_i. The *higher-order unification problem* is the problem of determining whether a disagreement set can be unified and, when it can be, of providing a unifier for it. We note that in the general case the question of existence of unifiers is only undecidable [Goldfarb, 1981]. Also, when unifiers do exist, there may not be a most general unifier. Nevertheless, a systematic search can be made for unifiers which succeeds in discovering them whenever they exist. We outline, with a small modification, the procedure in [Huet, 1975] which conducts such a search.

Certain disagreement sets, called *solved sets* here, have trivial unifiers (although computing all their unifiers can be quite hard). Certain other disagreement sets, called *failed sets* here, are easily seen to have no unifiers. The search for a unifier proceeds by attempting to reduce a given disagreement set to either a solved set or a failed set. Central to this process are the operations SIMPL, TRIV and MATCH. SIMPL attempts to simplify a disagreement set by looking at pairs of terms whose heads cannot be changed by substitutions. It either decides that the terms of at least one such pair cannot be unified, or reduces the question of unification of the terms of each such pair to that of the unification of their arguments. In the first-order case, this corresponds to descending through the pair of terms simultaneously so long as no variables are encountered and the term structures are identical at the top. Given a disagreement set $\mathcal{D}$, SIMPL returns the marker $\mathcal{F}$ if it has determined that $\mathcal{D}$ has no unifiers, or it produces a *simplified* disagreement set $\mathcal{D}'$ that has the same set of unifiers as $\mathcal{D}$. If $\mathcal{D}'$ is not a solved set then substitutions are necessary to continue the reduction process. TRIV examines a simplified disagreement set and returns the set of pairs in it of the form $\langle x, t\rangle$ where x is a variable and t is a term in which x does not appear free. (An implementation of TRIV may, of course, drop this "occur-check" condition, trading soundness with efficiency.) If there are such pairs, then any one of them may be used as a substitution to simplify the disagreement set further. SIMPL and TRIV are used repeatedly till either the set has been successfully reduced to a solved or failed set, or no further simplifications are possible. In the latter case strictly higher-order considerations are needed to carry the search process forward. This is the domain of the MATCH procedure. When MATCH is applied to a simplified disagreement set, it first picks a pair in the set and then produces a finite set of substitutions that help in unifying that disagreement pair. MATCH is therefore a non-deterministic function, since the value it returns depends of the choice of disagreement pair. We do not describe the structure of MATCH here due to a lack of space.

We may now define a notion of derivation relative to a set of definite clauses P. Let us use, perhaps with subscripts, the symbols $\mathcal{G}$ to denote a finite subset of $\mathcal{H}^+$, $\mathcal{D}$ to denote a disagreement set and θ to denote a substitution. Then the triple $\langle \mathcal{G}_2, \mathcal{D}_2, \theta_2\rangle$ is said to be *P-derived* from the triple $\langle \mathcal{G}_1, \mathcal{D}_1, \theta_1\rangle$ if the former is obtained from the latter by one of the following steps; in this definition, we say that a variable is *new* if it is does not occur free in any of the formulas that appear in $\langle \mathcal{G}_1, \mathcal{D}_1, \theta_1\rangle$.

(1) *Goal reduction and backchaining steps:* Let G be some member of $\mathcal{G}$ and let $\mathcal{G}'$ be the result of removing that occurrence of G from $\mathcal{G}$. In the first four cases below, set $\mathcal{D}_2 := \mathcal{D}_1$ and $\theta_2 := \emptyset$. We refer to the first five cases as *goal reduction* steps and the last one as the *backchaining* step.

(a) If G is *true*, then $\mathcal{G}_2 := \mathcal{G}'$.

(b) If G is $G_1 \wedge G_2$ then $\mathcal{G}_2 := \{G_1, G_2\} \cup \mathcal{G}'$.

(c) If G is $G_1 \vee G_2$, then $\mathcal{G}_2 := \{G_1\} \cup \mathcal{G}'$ or $\mathcal{G}_2 := \{G_2\} \cup \mathcal{G}'$.

(d) If G is ΣB, then $\mathcal{G}_2 := \{\lambda norm(By)\} \cup \mathcal{G}'$ for some new variable y.

(e) If G has a variable, y of type $\alpha_1 \rightarrow \ldots \rightarrow \alpha_n \rightarrow o$, as its head, then set $\theta_2 := \{\langle y, \lambda x_1 \ldots \lambda x_n true\rangle\}$, $\mathcal{G}_2 := \theta_2\mathcal{G}'$, and $\mathcal{D}_2 := \text{SIMPL}(\theta_2\mathcal{D}_1)$. Here, the type of x_i is α_i, for $i = 1 \ldots, n$.

(f) Otherwise, G has a nonlogical constant as its head. Let $\forall \bar{x}\ (G' \supset A)$ be a type variable free, type-instance of a clause in $\mathcal{P}$. Set $\theta_2 := \emptyset$, $\mathcal{G}_2 := \{G'\} \cup \mathcal{G}'$, and $\mathcal{D}_2 := \mathcal{D}_1 \cup \text{SIMPL}(\{\langle G, A\rangle\})$. Here we assume that the variables $\bar{x}$ are new.

(2) *Unification step:* If $\mathcal{D}_1$ is neither $\mathcal{F}$ nor a solved set, then we either apply TRIV or MATCH.

(a) If $\text{TRIV}(\mathcal{D}_1) \neq \emptyset$ then for any $\sigma \in \text{TRIV}(\mathcal{D}_1)$ set $\theta_2 := \sigma$, $\mathcal{G}_2 := \sigma\mathcal{G}_1$ and $\mathcal{D}_2 := \text{SIMPL}(\sigma\mathcal{D}_1)$.

(b) Let Θ be some value returned for $MATCH(\mathcal{D}_1)$. If Θ is empty, then $\mathcal{D}_1$ is recognized as a failed set. In this case, set $\mathcal{G}_2 := \mathcal{G}_1$, $\mathcal{D}_2 := \mathcal{F}$, and $\theta := \emptyset$. Otherwise, pick $\sigma \in \Theta$, and set $\theta_2 := \sigma$, $\mathcal{G}_2 := \sigma\mathcal{G}_1$ and $\mathcal{D}_2 := \text{SIMPL}(\sigma\mathcal{D}_1)$.

A sequence $\langle \mathcal{G}_i, \mathcal{D}_i, \theta_i\rangle_{i \leq n}$ in which, for each $i < n$, $\langle \mathcal{G}_{i+1}, \mathcal{D}_{i+1}, \theta_{i+1}\rangle$ is $\mathcal{P}$-derived from $\langle \mathcal{G}_i, \mathcal{D}_i, \theta_i\rangle$, is called a *$\mathcal{P}$-derivation* sequence. In addition, if $\mathcal{D}_0 = \emptyset$, $\theta_0 = \emptyset$ and $\mathcal{G}_0 = \{G\}$ then the sequence is said to be a $\mathcal{P}$-derivation sequence for G. Notice that sequences for which $\mathcal{G}_n = \emptyset$ and $\mathcal{D}_n$ is either a solved set or $\mathcal{F}$, cannot be extended. If $\mathcal{G}_n = \emptyset$ and $\mathcal{D}_n$ is a solved set, we say that that $\mathcal{P}$-derivation of G is a *proof of G from $\mathcal{P}$*, and that the substitution $\theta_n \circ \cdots \circ \theta_1$ is its *answer substitution.* The following theorem establishes the soundness and completeness for this notion of proof.

Theorem 3: Let $\exists \bar{x} G$ be a closed goal formula which contains no type variables. $\mathcal{P} \vdash_{\mathcal{T}} \exists \bar{x} G$ if and only if there is a $\mathcal{P}$-derivation sequence which is a proof of G from $\mathcal{P}$. In the latter case, if θ is the answer substitution for the sequence and σ is a unifier for the final solved set, then $\mathcal{P} \vdash_{\mathcal{T}} G'$ for every ground instance G' of $\sigma \circ \theta\ G$.

Notice that if $\mathcal{P}$ and G are essentially first-order, the final solved set of a proof of G from $\mathcal{P}$ is always empty, so σ can be taken to be the empty substitution. In this case, the notion of an answer substitution coincides with the usual (first-order) definition.

The mechanism that we desired at the beginning of this section may be described as one that starts with the triple $\langle \{G\}, \emptyset, \emptyset\rangle$ and performs an exhaustive search for a proof of G from $\mathcal{P}$. There are several choices in extending a derivation sequence, but most of these are inconsequential. A complete procedure may for instance choose to do any one of the unification steps or a backchaining step or one of the goal reduction steps 1(a)-1(d). Within the unification step 2(b), however, the choice of substitution may be critical. A similar observation applies to the choice of clause in 1(f).

In constructing an interpreter for λProlog, it appears inappropriate to perform a breadth-first search even where necessary, and trade-offs need to be made between completeness and practicality. We have designed an interpreter that performs a depth-first search with backtracking that

is similar to the one standard Prolog interpreters perform: It always chooses to do a unification step, applying TRIV, whenever possible. When a choice of goal has to be made it picks the first in the list. In determining a clause to use (1(f)), it picks the first appropriate one in a predetermined ordering. However there are a few points peculiar to our language that bear mentioning: (1) Before using 1(e) to solve a goal with a variable, y, as its head it is necessary for completeness to check that y does not appear free in an argument of any of the other goals or in the associated disagreement set. Our interpreter does not perform such a check, preferring instead not to reorder goals in the goal list. (2) Even after a clause has been chosen in 1(f), it is still necessary to choose a type instance of it. Our solution to this problem is to permit type variables in goals and to delay their determination until term unification. SIMPL and TRIV can be easily modified to deal with such variables, but there are problems in adapting MATCH to deal with type variables that need to be instantiated to function types. When it encounters such a case, our interpreter gives up and indicates a run-time error. A better analysis of this problem is clearly necessary, and must be based on a stronger formalization of type quantification. (3) Choices may have to be made in the unification step, and backtracking points need to be maintained for these as well. Our interpreter saves such points and can backtrack over them. We have implemented no control primitives for the unification search process. Although such controls will most certainly be necessary for various kinds of programs, we have been successful at running many λProlog programs which make non-trivial uses of higher-order unification without such controls.

There are several other issues pertaining to the interpreter that need to be discussed, but we omit these here due to a limitation on space.

Section 6: Conclusions

In this paper we have investigated the issue of introducing higher-order objects into a logic programming language. Toward this end we have used a higher-order logic to generalize the first-order Horn clauses. Our theoretical results show that this generalisation preserves certain important computational properties. We have described a programming language that is based on these results, and we have also outlined an interpreter for this language. Our current implementation of an interpreter was not designed with efficiency in mind. We are now investigating the design of an abstract machine to support a more efficient implementation.

The language that we have presented here gives first-class logical status to typed λ-terms of all types, and this constitutes a considerable enrichment to the data structures of Prolog. This enrichment brings with it a cost, viz a branching in unification, that at first sight may appear prohibitive. However, there are certain points to be noted. First of all, branching occurs only in cases that involve genuinely higher-order unification, and in these cases the cost may not be unacceptable. Moreover there are several uses of λ-terms where the unification involves no branching at all. Examples in this category include all the uses of Lisp-like features described in [D. H. Warren, 1982] and situations where the λ-terms are used solely for the purpose of performing computations through reductions. In cases like these the language described here provides a clear and theoretically well-understood implementation. Finally, our preliminary investigations indicate that an interpreter for λProlog may be written in such a way that it performs very efficiently for the first-order fragment without jeopardizing its ability to deal with higher-order terms. If this is indeed true, then the new additions to the language would be achieved in a manner that is strictly conservative.

Section 7: References

[Apt and van Emden, 1982] Krzysztof R. Apt and M. H. van Emden, "Contributions to the Theory of Logic Programming" *JACM*, Vol 29 (1982), 841 – 862.

[Church, 1940] Alonzo Church, "A Formulation of the Simple Theory of Types," Journal of Symbolic Logic **5** (1940), 56 – 68.

[Fortune, Leivant and O'Donnell, 1983] Steven Fortune, Daniel Leivant, and Michael O'Donnell, "The Expressiveness of Simple and Second-Order Type Structures", J.ACM Vol. 30(1), January 1983, pp. 151-185.

[Goldfarb, 1981] Warren D. Goldfarb, "The Undecidability of the Second-Order Unification Problem," *Theoretical Computer Science* **13** (1981), 225 – 230.

[Huet, 1975] Gérard P. Huet, "A Unification Algorithm for Typed λ-calculus," Theoretical Computer Science **1** (1975), 27 – 57.

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

[Miller, 1983] Dale A. Miller, "Proofs in Higher-order Logic," Ph. D. Dissertation, Carnegie-Mellon University, August 1983.

[Miller, 1986] Dale A. Miller, "A Theory of Modules for Logic Programming," University of Pennsylvania Techincal Report, 1986.

[Miller and Nadathur, 1985] Dale A. Miller, Gopalan Nadathur, "A Computational Logic Approach to Syntax and Semantics," 10^{th} Annual Symposium of the Mathematical Foundations of Computer Science, IBM Japan, May 1985.

[Miller and Nadathur, 1986] Dale A. Miller, Gopalan Nadathur, "An Abstract Interpreter for a Higher Order Extension of Prolog," forthcoming UPenn technical report, December 1985.

[Milner, 1978] Robin Milner, "A Theory of Type Polymorphism in Programming," *Journal of Computer and System Sciences* **17**, 348 – 375, 1978.

[Mycroft and O'Keefe, 1985] A. Mycroft and R. A. O'Keefe, "A Polymorphic Type System for Prolog," *Artificial Intelligence*, Vol. 23(3), August 1984.

[Reynolds, 1985] J. C. Reynolds, "Three Approaches to Type Structure", Proceedings of the International Joint Conference on Theory and Practice of Software Development, March 1985.

[van Emden and Kowalski, 1976] M. H. van Emden, R. A. Kowalski, "The semantics of predicate logic as a programming language," J.ACM 23, 4 (Oct. 1976), 733 – 742.

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

[D. S. Warren, 1983] David Scott Warren, "Using λ-Calculus to Represent Meaning in Logic Grammars" in the Proceedings of the 21st Annual Meeting of the Association for Computational Linguistics, June 1983, 51 – 56.