AN OVERVIEW OF λPROLOG

GOPALAN NADATHUR
Department of Computer Science
Duke University
Durham, NC 27706

DALE MILLER
Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104-6389

Abstract: λProlog is a logic programming language that extends Prolog by incorporating notions of higher-order functions, λ-terms, higher-order unification, polymorphic types, and mechanisms for building modules and secure abstract data types. These new features are provided in a principled fashion by extending the classical first-order theory of Horn clauses to the intuitionistic higher-order theory of *hereditary Harrop* formulas. The justification for considering this extension a satisfactory logic programming language is provided through the proof-theoretic notion of a *uniform proof.* The correspondence between each extension to Prolog and the new features in the stronger logical theory is discussed. Also discussed are various aspects of an experimental implementation of λProlog.

Appears in the Fifth International Conference Symposium on Logic Programming, 15 – 19 August 1988, Seattle, Washington. This is a slightly corrected version of the paper in the Proceedings.

Address correspondence to gopalan@cs.duke.edu or dale@linc.cis.upenn.edu or to the addresses above.

1. Introduction

The logic programming language λProlog is an extension of conventional Prolog [32] in several different directions: it supports higher-order programming, incorporates λ-terms as data structures, includes a notion of polymorphic typing, and provides mechanisms for defining modules and secure abstract data types. There have been several proposals in the past for adding features of this sort to Prolog. The work in the context of λProlog is distinguishable from most of these proposals in that the first concern has been to examine the essential logical and proof-theoretic nature of these extensions. The result of this analysis has been the description of a class of formulas that are called *higher-order hereditary Harrop* (hohh) formulas. These formulas play a role in λProlog that is similar to the role played by first-order positive Horn clauses in Prolog. The hohh formulas significantly extend positive Horn clauses, and the new features provided in λProlog are the result of exploiting the extension found in hohh formulas.

We discuss several aspects of the work on λProlog in this paper. In the next section we describe the hohh formulas and provide the rationale for considering them a suitable basis for a logic programming language. Section 3 highlights the logical features that are new in hohh formulas and explains their use in providing extensions to logic programming. Finally, we describe an implementation of a version of this logic in Section 4, dwelling on some of the new problems that were encountered in its context.

2. Reconsidering the Foundation in Horn Clauses

We initially considered a higher-order extension to Horn clauses because we were interested in addressing an aspect of incompleteness in theorem provers in higher-order logic. In logics that permit predicate quantification, new techniques must be devised for finding appropriate substitutions for predicates variables since these cannot in general be determined through standard uses of (even higher-order) unification. This problem appeared to be very difficult to solve for general higher-order logic [1], and it therefore seemed natural to focus attention on a sublogic. One possibility was to consider some form of higher-order extension to Horn clauses. This possibility appeared to be independently interesting since it also provided a basis for studying higher-order notions in logic programming, an aspect for which an analysis was missing from the literature.

As it turned out, we succeeded in solving the problem of predicate substitutions for higher-order Horn clauses [26]. This solution did not shed much light on the general theorem proving problem: the Horn clause setting is so weak that the needed predicate substitutions can be determined solely through higher-order unification. However, our results did lead to an understanding of the properties of higher-order Horn clauses and established arguments for their suitability as a basis for logic programming. Using these results, we described the first version of λProlog and constructed an interpreter for it [21]. This interpreter extended the standard interpreter for Prolog essentially by replacing first-order unification with higher-order unification.

In our analysis of higher-order Horn clauses, we represented proofs using the sequent calculus [4, 31] instead of the more traditional format of resolution refutations. This turned out to be rather fortunate because we were able to observe certain structural properties of sequent proofs involving positive Horn clauses that appear to capture the proof-theoretic "essence" of logic programming. After these properties were abstracted out, it was natural to look for extensions to positive Horn clauses that also satisfied them. One such extension involving the addition of implications to the body of definite clauses was described in [16, 18]. A further extension that permitted universal quantification in the body of definite clauses was described in [17, 23]. This final extension is what we refer to as *hereditary Harrop* formulas.

Our criterion for judging the adequacy of a logical theory as the basis for logic programming may be explained as follows: the theory should permit the logical connectives to be construed as certain simple search instructions while at the same time being true to their declarative intent. Consider, for instance, the logic underlying Prolog. It is possible to construct a complete proof procedure for this logic that interprets the connectives $\wedge$ and $\vee$ in a query or the body of a program clause as specifications of AND and OR nodes in a search space. This aspect permits a programmer writing in this logic to understand clearly the nature of the computation being described. In contrast, the connection between logical connectives and the search that needs to be performed appears to be much too complex in general logic for this to be considered a medium for programming.

2. Reconsidering the Foundation in Horn Clauses

The relation between the logical connectives $\vee$ and $\wedge$ and the search operations OR and AND can be generalized to other logical connectives, in particular, implication ($\supset$), and universal ($\forall$) and existential ($\exists$) quantifiers. The sequent calculus provides the means to establish a correspondence between the declarative meaning of these logical connectives and search operations. When the succedent introduction rule for a particular logical connective is read in a top-down manner, it supplies the declarative meaning for that connective. Viewing logic programming as a process for *constructing* sequential proofs in a bottom-up fashion, the backward reading of the introduction rule supplies the natural search-related meaning for the logical connective. Adding new logical connectives in this manner actually results in the addition of new search primitives to logic programming, in contrast to the approach taken by Lloyd and Topor [13] where the new connectives are removed by using equivalences in classical logic and by interpreting negation with negation-by-failure.

The association between logical connectives and search operations is defined precisely in [24]. This definition is a refinement of the one in [23] and can be summarized as follows. Let $\mathcal{L}$ be a formulation of quantificational logic containing at least the logical connectives $\wedge$, $\vee$, $\supset$, $\forall$, and $\exists$. Let $\vdash$ be any provability relation over the formulas in $\mathcal{L}$ that has a formulation in terms of a sequent calculus. In this context, we first define a *uniform proof* to be a cut-free sequential proof in which the succedent of each sequent is a single formula and, further, every sequent whose succedent is a non-atomic formula is the lower sequent of an inference figure that introduces its top-level connective. Given any two sets of $\mathcal{L}$-formulas, $\mathcal{D}$ and $\mathcal{G}$, we then say that $\vdash$ is *uniform* over the pair $\langle \mathcal{D}, \mathcal{G} \rangle$ if for every finite subset $\mathcal{P}$ of $\mathcal{D}$ and every $G \in \mathcal{G}$ there is a uniform proof of G from $\mathcal{P}$ whenever it is the case that $\mathcal{P} \vdash G$. Members of $\mathcal{D}$ are called *definite clauses* or *program clauses* while members of $\mathcal{G}$ are called *goals* or *queries*.

From this definition it follows that if $\vdash$ is uniform over $\langle \mathcal{D}, \mathcal{G} \rangle$ and $\mathcal{P} \vdash G$ where $G \in \mathcal{G}$ and $\mathcal{P}$ is a finite subset of $\mathcal{D}$, then the following conditions hold:

AND If G is $B \wedge C$ then $\mathcal{P} \vdash B$ and $\mathcal{P} \vdash C$.

OR If G is $B \vee C$ then $\mathcal{P} \vdash B$ or $\mathcal{P} \vdash C$.

AUGMENT If G is $B \supset C$ then $B, \mathcal{P} \vdash C$.

INSTANCE If G is $\exists x\ B$ then there exists a term t such that $\mathcal{P} \vdash [t/x]B$.

GENERIC If G is $\forall x\ B$ then $\mathcal{P} \vdash [c/x]B$ where c is a constant that does not appear in $\mathcal{P}$ or G.

These conditions provide what we consider to be the search related interpretation of the various connectives. Using these notions, an *abstract logic programming language* is defined to be a quadruple $\langle \mathcal{L}, \mathcal{D}, \mathcal{G}, \vdash \rangle$ where $\mathcal{D}$ and $\mathcal{G}$ are sets of $\mathcal{L}$-formulas such that $\vdash$ is uniform over $\langle \mathcal{D}, \mathcal{G} \rangle$.

One example of an abstract logic programming language is provided by the logic of positive Horn clauses. To be precise, let $\mathcal{F}$ be a formulation of first-order logic and let $\vdash_C$ denote classical provability. Let $\mathcal{G}_1$ be the set of all first-order formulas that are the existential closure of formulas of the form $A_1 \wedge \ldots \wedge A_n$, where $n > 0$ and the A's are atomic. Finally, let $\mathcal{D}_1$ be the set of all first-order formulas that are the universal closure of formulas of either the form $[A_1 \wedge \ldots \wedge A_n] \supset A_0$ where $n > 0$ or the form A_0: here again, the A's are atomic. Members of $\mathcal{D}_1$ are often referred to as *positive Horn clauses*, while the negation of member of $\mathcal{G}_1$ are often referred to as *negative Horn clauses*. It can then be shown that $\langle \mathcal{F}, \mathcal{D}_1, \mathcal{G}_1, \vdash_C \rangle$ is an abstract logic programming language. It should be noted, however, that this is a weak abstract logic programming language in the sense that the search operations OR, AUGMENT, INSTANCE, and GENERIC are never used. A language which uses the OR and INSTANCE search operations can be obtained easily by adding disjunction and existential quantification to formulas in $\mathcal{G}_1$ and the body of clauses in $\mathcal{D}_1$.

As undertaken in [21] and [26], a higher-order extension to the logic of positive Horn clauses can be described and shown to be an abstract logic programming language. The latter extension provides many new programming features through the availability of λ-terms and higher-order unification. However, this language is still weak in its use of logical connectives since it does not incorporate the AUGMENT or the GENERIC search operations.

Including the AUGMENT operation requires a shift from the framework of classical logic. Examining simple extensions that permit implications in goals reveals why this must be so. Consider, for instance, the formula $p \vee (p \supset q)$. Although this formula has a proof in classical logic, it has no proof that is consistent with the OR interpretation of $\vee$: neither p nor $p \supset q$ is provable. As another example, consider the positive Horn clause $p(a) \wedge p(b) \supset q$ and the "goal" $\exists x(p(x) \supset q)$. While this goal has a classical proof from the given Horn clause, there is no proof that is consistent with the INSTANCE interpretation of $\exists$; there is no single instance of the goal that is provable from the given Horn clause. Intuitionistic logic can, however, provide a declarative meaning of implication which is consistent with the AUGMENT search operation. Intuitionistic provability, denoted here by $\vdash_I$, is weaker than classical provability, and this weakening enables it to be used to define stronger abstract logic programming languages. This statement is not as paradoxical as it may sound since the abstract logic programming languages $\langle \mathcal{F}, \mathcal{D}_1, \mathcal{G}_1, \vdash_C \rangle$ and $\langle \mathcal{F}, \mathcal{D}_1, \mathcal{G}_1, \vdash_I \rangle$ are essentially the same. That is, intuitionistic logic provides a "tighter" analysis of Horn clauses than classical logic does.

Consider the class of first-order formulas denoted by the following mutually recursive definition of the syntactic variables D and G:

$$D := A \mid G \supset A \mid \forall x\, D \mid D_1 \wedge D_2$$

$$G := A \mid G_1 \wedge G_2 \mid G_1 \vee G_2 \mid \forall v\, G \mid \exists v\, G \mid D \supset G.$$

We assume here that A ranges over atomic formulas. A D formula is called a *hereditary Harrop* formula. Let $\mathcal{D}_2$ be the set of hereditary Harrop formulas and let $\mathcal{G}_2$ be the set of G-formulas. Using methods described in [18], it can be shown that $\langle \mathcal{F}, \mathcal{D}_2, \mathcal{G}_2, \vdash_I \rangle$ is an abstract logic programming language. It is, in fact, a language in which all the search operations are present. The term "hereditary Harrop formulas" is used since these formulas and all their positive subformulas satisfy a syntactic restriction described by Harrop [9]: formulas in $\mathcal{D}_2$ are essentially equivalent to those formula that do not contain any positive occurrences of disjunctions or existential quantifiers.

Finally, it is possible to define a higher-order version of hereditary Harrop formulas. The top-level structure of these formulas is similar to that of the first-order ones. One difference is that in the former case quantification is permitted over variables of all orders and is restricted only in that the top-level symbols of the formulas denoted by A in the definition of D formulas must be nonlogical constants. Furthermore, logical constants may appear embedded in terms in the higher-order versions of both Horn clauses and hereditary Harrop formulas. In the former case these constants are limited to $\wedge, \vee, \exists$, and in the latter case they are limited to $\wedge, \vee, \exists, \forall$. See [24] for more details.

3. Programming Language Consequences

Higher-order hereditary Harrop formulas extend Horn clauses in two major respects: first, they embed higher-order notions and, second, they provide the primitives for specifying the additional search operations AUGMENT and GENERIC. These two extensions to the logic are orthogonal in the sense that they can each be added independently of the other. They are, however, related from a programming point-of-view in that they both provide the means for realizing certain abstraction mechanisms that are routinely found in modern programming languages but are not directly obtainable through the use of first-order Horn clauses. In this respect, λ-terms and predicate variables provide for notions of higher-order programming; AUGMENT can be used to realize a notion of modules; GENERIC provides a mechanism for realizing abstract data types. The presence of λ-terms also leads to a feature that is novel to the programming realm: they can be used as data structures for representing objects containing variable bindings. We discuss these logical extensions in greater detail below and explain how they lend themselves to the addition of new features at a programming level.

Predicate Variables. Functional programming languages such as Lisp and ML provide the ability to write functions that can take other functions as arguments. Since the logic programming correspondent to a function is a predicate, the provision of this feature in logic programming is related to the ability to quantify over predicates. Such quantification is permitted in hohh formulas and, consequently, several aspects of this kind of higher-order programming can be realized by using these formulas. For instance, the hohh formulas (written in a notation borrowed from Prolog)

$$
\begin{array}{l}
mappred(P, [\,], [\,]). \\
mappred(P, [X|L1], [Y|L2]) \;\; \texttt{:-} \;\; P(X, Y), mappred(P, L1, L2).
\end{array}
$$

define a predicate *mappred* that corresponds to the function *maplist* in Lisp; these formulas are higher-order ones because they contain a quantification over a predicate variable P. Languages like Lisp also permit arbitrary lambda expressions to be passed as arguments and later evaluated as programs. Limited aspects of this feature are provided by hohh formulas as well, since λ-terms containing embedded logical connectives may appear as arguments of atomic goal formulas. For example, the predicate *mappred* above can be "invoked" with its predicate argument instantiated by a complex goal.

λ-terms as Data Structures. A logic programming language based on hohh formulas must provide simply typed λ-terms as its data structures with the equality between these terms being given by λ-conversion. These data structures turn out to be especially convenient in programming tasks involving the manipulations of objects containing variable bindings. Although first-order terms can be used to represent objects that contain internal abstractions, such representations make it necessary for the programmer to deal with variable names and to implement the various substitution operations that would be needed. In contrast, λ-terms capture the *higher-order abstract syntax* of these objects [30]. Using them as representational devices makes the handling of bound variable names, substitutions, variable capturing, *etc.*, part of the meta theory of the programming language. Consequently the mechanisms for these are directly supported by the implementation of the underlying language.

As an illustration, consider formulas of a quantificational logic. For instance, the formula $\forall x[p(x) \vee q(x)]$ can be represented by the λ-term $(all\ \lambda x((p\ x)\ or\ (q\ x))$, where *all* and *or* are constants of the appropriate types: in the case of *all*, it has a higher-order type, that is, it maps abstractions over booleans to booleans. The advantage of this representation is that several useful operations on formulas are supported directly by λ-conversion. Thus, the equivalence of this formula to $\forall y[p(y) \vee q(y)]$ is mirrored in the α-convertibility of the two terms $(all\ \lambda x((p\ x)\ or\ (q\ x))$ and $(all\ \lambda y((p\ y)\ or\ (q\ y))$. Similarly, instantiating the given formula with the term $f(a)$ is simply expressed by the term $(\lambda x((p\ x)\ or\ (q\ x))(f\ a)$; this term is equal (modulo λ-conversion) to the representation of the desired instance, $(p\ (f\ a))\ or\ (q\ (f\ a))$. The richer notion of equality also makes it possible to perform quite sophisticated pattern matching operations. For instance, the term $(all\ \lambda x((P\ x)\ or\ (Q\ x)))$ in which P and Q are (higher-order) variables can be used as a template for recognizing formulas containing a top-level universal quantifier whose scope is a disjunction. Observations such as these have in fact been exploited to provide succinct implementations of theorem-provers, interpreters, and type inference procedures [2, 20, 22, 27, 29].

As another example, λ-terms turn out to be particularly apt for representing programs that contain formal parameters or local variables. Arguments similar to those above show that a logic programming language incorporating such terms as well as higher-order unification provides useful features for the implementation of program manipulating procedures. See [7, 8, 11, 22] for examples of building such systems using higher-order logic.

A consequence of incorporating the rules of λ-conversion is that the interpreter for the logic of hohh formulas must be able to solve equations based on these rules. Formally, this means building higher-order unification into the interpreter. This form of unification is considerably more complex than first-order unification: it is undecidable in general and most general unifiers may not exist even when unifiers do exist [10]. However, there are several characteristics that a search for a unifier shares with the search for a proof of goal from first-order Horn clauses, and it is possible to interleave these two searches in designing an interpreter for the language under consideration [21, 26]. It is exactly this kind of an interpreter that is used in the current version of λProlog, as we discuss briefly in the next section. Meanwhile, we note that despite the complexity of higher-order unification, the kinds of unification problems that have arisen in applications so far have been tractable and have had correspondences to conceptual problems that a trained λProlog programmer can easily recognize.

AUGMENT. This search operation provides logic programming with an aspect of hypothetical reasoning [3, 14, 15]. It can also be used to realize a mechanism for supporting modular programming [16, 18]. To achieve the latter, implications can be used to structure, in a stack disciplined fashion, the environments within which queries are to be attempted. Consider, for instance, the query

$$(D_1 \supset (G_1 \wedge (D_2 \supset G_2))) \wedge G_3$$

in the context of the program $\mathcal{P}$. In solving this query, three different "environments" are used for solving the three goals G_1, G_2, and G_3: $\mathcal{P} \cup \{D_1\}$ for G_1, $\mathcal{P} \cup \{D_1, D_2\}$ for G_2 and $\mathcal{P}$ for G_3. By exploiting the possibility of nesting implications, a notion of "importing" modules of code can also be supported. It is important to note that the notion of modules implemented via AUGMENT is slightly different from that in conventional programming languages. In such languages, the meaning of a procedure defined in a module is determined in a completely local manner. In contrast, there is an "openness" in the context of modules described here that is in harmony with the interpretation of procedures in logic programming. For instance, if D_1 and D_2 are two modules that contain clauses defining the same procedure, then the meaning of this procedure in an environment that imports both D_1 and D_2 is obtained by combining the sets of clauses in these two modules.

GENERIC. This search operation attempts to verify a universal goal by trying to prove the result of instantiating the goal with a new object. This is actually an *intensional* interpretation of the universal quantifier that is to be contrasted with the *extensional* interpretation often used in Prolog systems; the latter interpretation attempts to satisfy a universal goal by trying to verify that it holds for every element in a given domain. For example, let the predicates $P(x)$ and $Q(x)$ stand for "x is a president of the USA" and "x is born in the USA" respectively. Then the intensional reading of $\forall x(P(x) \supset Q(x))$ corresponds to the question of whether a president of the USA must necessarily be born in the USA, whereas the extensional reading corresponds to the question of whether all the known presidents of the USA have in fact been born in the USA. Clearly the intensional interpretation is logically stronger since it implies the extensional interpretation. There are applications where both of these interpretation are useful.

The condition for "newness" in the GENERIC search operation can be used to provide a kind of security in unification. Consider, for example, the goal $\exists x \forall y\ P(x, y)$. To solve this, the interpreter might introduce a free variable, say X, for the top-level existential quantifier thus reducing the query to $\forall y\ P(X, y)$. At this stage, the interpreter must choose a constant, say c, that does not occur in the current program clauses and the goal, and then establish $P(X, c)$. In doing so it must ensure that X is at no stage instantiated to a term that contains c. This ability to restrict substitutions can be used to provide a *local* construct in program modules. For instance, the query $\exists x \forall y(D(x, y) \supset G(x))$ requires a substitution to be found for x that does not contain the parameter substituted for y. Assuming that G does not contain y free, it can be seen that $\exists x \forall y(D(x, y) \supset G(x))$ is equivalent to $\exists x(\exists y\ D(x,y) \supset G(x))$ in intuitionistic logic. Thus, constants that are declared to be local to a module can formally be thought of as variables that are existentially quantified over program clauses in the module. The mechanism for "hiding" these constants is then implemented directly by the GENERIC search operation.

Types. One other difference between hohh formulas and first-order Horn clauses is that the former are actually typed formulas. The essential role of types here is to impose a functional hierarchy on terms. From a programming perspective, these types provide an ML-like typing discipline [25] to Prolog. Such types add an element of documentation to programs, and type errors detected at parsing time identify goals that will never succeed. The typing scheme (which initially uses only primitive and functional types) can be embellished with type constructors. For instance, it is possible to define *list* as a type constructor that maps the type *int* to the new "primitive" type (*list int*) corresponding to lists of integers. Some form of polymorphic typing appears to be necessary for the convenient construction of typed programs, and so our current implementation of the logic permits variables as types. Such types, however, are not supported by the theory of hohh formulas and they turn out to be computationally expensive in certain instances; we discuss this issue in the next section.

4. Implementing the Extended Logic

The λProlog system in its current form is an implementation of, roughly, the logic of hohh formulas; the main deviations are that implications in goals is incompletely implemented and that a form of polymorphic typing has been incorporated. A version of this system comprising roughly 4100 lines of Prolog code has been in existence since August 1987 and has been distributed to about 40 sites in North America, Europe, and Asia. The performance of this system leaves much to be desired, partly because of the underlying implementation language and to a greater extent because efficiency has not been a major concern in this experimental implementation. Despite this drawback, the system has been used for serious experimentation and prototype implementations [2, 7, 8, 20, 22, 29] with two ongoing Ph.D. theses using it as their primary implementation vehicle. We describe this system briefly in this section, focussing on the solutions adopted to the new problems encountered in implementing the logic.

From the user's perspective, programming in λProlog consists largely of writing a collection of type declarations and program clauses. The following declaration, which identifies $int \rightarrow (list\ int) \rightarrow (list\ int)$ as the type of *cons*, illustrates the format of type declarations.

$$type \quad cons \quad int \rightarrow (list\ int) \rightarrow (list\ int).$$

In this type expression, *int* is intended as a primitive type and *list* as a type constructor, and there are provisions for the user to declare these as such. Type expressions can also contain variables: thus, the type of *cons* could also have been defined to be $A \rightarrow (list\ A) \rightarrow (list\ A)$, where A (and, in general, any token beginning with an upper-case letter) is a variable.

When writing program clauses in λProlog, we have retained the symbols of Prolog, embellishing these as needed by the richer syntax of the underlying logic. Several syntactic conventions have also been retained, with the exception that a curried notation has been adopted. This is exemplified by the two clauses below that define a predicate *mapfun* of three arguments:

$$mapfun\ F\ []\ [].$$
$$mapfun\ F\ [X|L1]\ [(F\ X)|L2]\ \texttt{:-}\ mapfun\ F\ L1\ L2.$$

The types of the symbols that appear in clauses can be defined by type declarations. They can also be inferred by techniques similar to those used in ML [5]. For instance, from the clauses above, the type of *mapfun* can be inferred to be $(A \rightarrow B) \rightarrow (list\ A) \rightarrow (list\ B) \rightarrow o$, where o is the type for propositions in λProlog.

Type declarations and program clauses can be organized into named collections called *modules*. Modules also provide a notion of scope that is useful in type inferencing: when the code in a module is being type checked and the type of a constant has not been declared, all occurrences of that constant in a module are assumed to be of the same type while all occurrences of that constant outside of the module are assumed to be instances of its inferred type. Modules can also be *imported* into other modules. From the perspective of program clauses, importing is explained by means of the AUGMENT search operation [16, 18]. In understanding it operationally, the notion of "current module list" is useful. Suppose the current module list contains M_1 and M_2 and that M_1 imports module M_3. Given an atomic goal, A, a search is made only in the current modules for a clause whose head unifies with A: the module M_3 is not accessed. Suppose such a clause is found in M_1. In attempting to solve the body of this clause, the current module list is extended with the modules imported into M_1, in this case, M_3. That is, clauses in M_3, M_1 and M_2, in that order, can be used in attempting to solve this body. The current implementation of λProlog does not support the notion of parametric modules found in [16, 18].

The interpreter for λProlog is essentially a procedure that attempts to construct proofs for goal formulas from given sets of hohh formulas. The search space that such a procedure must deal with can be characterized by graphs whose nodes are pairs, the first element of which is a list of goal formulas and the second element is a list of disagreement pairs. For the initial state, the goal set contains only the goal formula for which a proof has to be found and the set of disagreement pairs is empty. The objective, then, is to reduce the goal set to an empty set and the disagreement set to one for which unifiers can easily be provided. In order to "solve" a compound goal, the search operations corresponding to each logical connectives can be used with the following exception: the choice of instantiation in the case of INSTANCE is delayed by substituting a variable whose value may be determined later through unification. In solving an atomic goal A whose top-level predicate symbol is a constant, clauses in the current modules must be considered. Thus, an attempt to use such a clause, say A' `:-` G, would lead to the addition of the pair $\langle A, A' \rangle$ to the disagreement set. The goal set is changed by removing A and adding G, if this is present. The overall structure of the search is quite similar to that in the context of first-order definite clauses. There are, however, the following differences: it is higher-order unification that must now be used, and the unification process must carefully deal with the special constants introduced by the GENERIC search operation.

The interpreter for λProlog performs a depth-first backtracking search, always trying to solve any unification problem first by using the search procedure described in [10]. It is thus quite similar to the standard Prolog interpreter. One difference is that unification problems may now involve a branching search, and may therefore introduce backtracking points. For instance, assume that g is a constant and F is a free variable, and consider the task of solving the following goal given the definition of $mapfun$ earlier in this section:

$$mapfun\ F\ [1,1]\ [(g\ 1\ 1),(g\ 1\ 2)].$$

This gives rise to the problem of finding a unifier for the disagreement pair $\langle (F\ 1),(g\ 1\ 1)\rangle$. There are four incomparable substitutions for F, namely $\lambda x(g\ x\ x)$, $\lambda x(g\ x\ 1)$, $\lambda x(g\ 1\ x)$, and $\lambda x(g\ 1\ 1)$, that will unify these terms. Of these, only the third will also unify the pair $\langle (F\ 2),(g\ 1\ 2)\rangle$. Thus if the interpreter picks any other substitution, it would have to backtrack over this choice.

The behavior of the interpreter can be improved by having it recognize special kinds of unification problems. As an example, consider a disagreement pair of the form $\langle X, F\rangle$ where X is a variable and F is a term in which X does not appear free. As noted in [10], the substitution $\{\langle X, F\rangle\}$ is a most general unifier for this pair. This special case can be strengthened in the presence of the η-rule [26], and the this strengthened version is incorporated in our current interpreter. Similarly, there is a suitably modified version of the *occurs check* (see the discussion about *rigid paths* in [10]) that can in certain instances be used to determine non-unifiability of higher-order terms. Incorporating these two special cases into the interpreter has the effect of handling first-order unification in a complete fashion. Finally, there are certain disagreement sets, the *flexible-flexible* sets, for which unifiers are known to exist but attempting to compute them might lead to an extremely redundant search. Our interpreter retains these sets as constraints instead of attempting to solve them. If they persist to the end of a computation, they are printed out along with the answer substitution. These sets can often be simplified to the extent that the user can actually read them and understand the manner in which they constrain the answer substitution [19]. This aspect of unification is in some sense reminiscent of the approach used in Constraint Logic Programming [12].

As noted earlier, a straightforward implementation of GENERIC requires the unification procedure to be modified to deal with the special constants correctly. An alternative approach, incorporated in our interpreter, is to modify the implementation of INSTANCE and GENERIC so that the unification procedure can be used unchanged. To understand the mechanics of this approach, let us assume initially that the program consists of only the quantified atomic formula $\forall u\ Au$ and that the goal is $\exists x \forall y \exists z (G\ x\ y\ z)$, where G is a constant. Assuming that x, y, h, and f are variables that do not appear in A, it can be seen that the given goal has a solution if and only if the pair $\langle \lambda y.A\ (h\ y), \lambda y.G\ x\ y\ (f\ y)\rangle$ has a unifier. Further, for any given constant c it is possible to obtain two terms t and s from a unifier for this pair such that c does not appear in t and $(G\ t\ c\ s)$ is provable from $\forall u\ Au$. Generalizing on this observation, the interpreter behaves as follows. It maintains a list of "universal variables" as part of its state. This list is initially empty. Whenever a universally quantified goal is encountered, the quantifier is dropped from the goal and the variable is added to the list; renaming is necessary here if the variable already appears in the list. When an existential quantifier is encountered in the goal, it is instantiated by a term of the form $(f\ x_1 \ldots x_n)$ where f is a new variable and the list of universal variables is $x_1, \ldots, x_n$. Finally, assume that an atomic goal A is encountered, with the list of universal variables once again being $x_1, \ldots, x_n$. To determine whether a particular clause in the program can be backchained upon in order to solve this goal, an instance of the clause is created by replacing its universal variables with terms of the form $(f\ x_1 \ldots x_n)$, where f is a new variable. Let A' be the head of the resulting instance, and let G be the body. An attempt is then made to solve the disagreement pair $\langle \lambda x_1 \ldots \lambda x_n A, \lambda x_1 \ldots \lambda x_n A'\rangle$. If this succeeds, the resulting unifier is used to determine the instance of G that must be solved next.

The approach to dealing with universal quantification outlined above is similar to a technique called $\forall$-lifting in [28]. There is an alternative scheme that essentially preserves the naive implementation of INSTANCE and GENERIC, and modifies the unification process instead. The basic idea here is to use an environment that remembers the scope of various quantifiers. This environment can then be used within the unification procedure of [10] to eliminate illegitimate substitutions; an analysis of how this might be done appears in [19]. Nadathur and Pfenning have implemented a version of λProlog that uses this approach to GENERIC. There appear to be certain computational payoffs in using this approach.

There are certain issues that need to be dealt with in a more complete fashion than is done in the current implementation of λProlog. The first of these concerns the provision of programming primitives for controlling the unification process. Some preliminary devices for this are incorporated in our interpreter [26], but there is room for much more experimentation in this respect. Another aspect concerns the polymorphic typing scheme that we have included in the language. This feature provides several programming conveniences that have been discussed in the context of functional programming languages. As one example, it permits the clauses for $mapfun$ provided earlier in this section to define a generic procedure that can be used by instantiating the variable F using function terms of several different types. Unlike the functional programming context, types in λProlog play an important role at runtime since they are referred to by higher-order unification. As a result, instantiations for type variables often need to be determined at run-time. There are techniques that can be used for either determining these instantiations uniquely or for delaying their determination in the hope that the desired instantiation can later become apparent [26]. However, there are times when none of these techniques apply and only fully enumerating all type instantiations is complete. Our interpreter does not perform such an enumeration, preferring to indicate a run-time error instead. This is clearly unsatisfactory.

5. Conclusion

This paper has discussed the design philosophy, the possible applications, and some aspects of an implementation of the λProlog system. The design of this system has been based entirely on proof-theoretical considerations. Also, the theory of hereditary Harrop formulas contains several features that are found in logics proposed by other researchers. The AUGMENT search operation is considered by Gabbay because it helps capture some aspects of the meta theory of Horn clauses [3]. Instances of both AUGMENT and GENERIC are added to Horn clauses by McCarty to provide inference mechanisms needed in certain AI reasoning tasks [14, 15]. Hallnäs and Schroeder-Heister generalize Horn clause logic with the ability to assume and discharge Horn clauses [6], and embedded implications overlap in functionality with their "rules of higher-level". Finally, Paulson's theorem prover Isabelle specifies inference rules using a subset of higher-order hereditary Harrop formulas. Isabelle contains not only higher-order unification but also operations closely related to the AUGMENT and GENERIC search operations [28].

5. Conclusion

The focus on proof-theoretic techniques in our study of logic programming has led to our ignoring programming aspects not captured directly by proof theory. These include notions of control, negation-by-failure, and side-effects. Clearly these issues are important in a practical programming language, and future theoretical work needs to address them.

Much work also needs to be done towards providing a serious implementation of either λProlog or a language that captures some of its extended functionality. One major task in this regard is to find a satisfactory representation for λ-terms. Such a representation must be one that enables an easy examination of the structure of these terms and also permits λ-conversion to be done efficiently. Further, it should be possible to undo the effects of λ-conversions rapidly; this is necessitated by the fundamental role of backtracking in the logic programming paradigm, and the fact that the application of a substitution in our context corresponds to λ-conversion. An important question to be answered is whether an interpreter for λProlog can be made to be as efficient on first-order Horn clauses as standard Prolog interpreters. Our current implementation is unfortunately too naive to shed any light on this question.

Along a different direction, there is a need to remove the mystery surrounding higher-order unification. Initially, it was very difficult for us to write programs that made significant use of this operation: it is a very different way of thinking about computations. With some practice it has become an easy matter for us to discern when we can gain from using this operation. However, there is as yet no general account of "computing with higher-order unification", our experiences and those of our colleagues being contained mainly in a list of example programs. Providing such an account so that programmers can learn when to use this operation appears to be worthwhile. Another aspect, which we are only beginning to understand, is that of knowing when a simple depth-first interpreter for unification will terminate and with how many different unifiers. Most of the unification problems that we have encountered in practice are solvable by this simple interpreter, but we have not yet formalized this experimental observation.

Finally there is the issue, raised in the last section, of how to deal with the interplay of higher-order unification and polymorphic types. It appears necessary to consider an analogue of hereditary Harrop formulas within a logic that contains explicit type quantification in order to address this problem adequately.

6. Acknowledgements

Our efforts in implementing λProlog have received much encouragement from the interest that Frank Pfenning has expressed in it from the beginning. We are also grateful to him, Conal Elliott, Amy Felty, Elsa Gunter, and John Hannan for being patient and helpful users of various versions of this system. Their comments have been extremely useful in the updates leading to Version 2.6. Nadathur's work has been partially supported by a Burroughs Contract. Miller is supported in part by NSF grant CCR-87-05596 and DARPA N000-14-85-K-0018.

7. References

[1] P. Andrews, D. Miller, E. Cohen, and F. Pfenning, Automating Higher Order Logic. In *Automated Theorem Proving: After 25 Years*, ed. W. W. Bledsoe and D. W. Loveland, AMS Contemporary Mathematical Series, Vol. 29, 1984.

[2] A. Felty and D. Miller, Specifying Theorem Provers in a Higher-Order Logic Programming Language. Ninth International Conference on Automated Deduction, Argonne Ill, May 1988, 61 – 80.

[3] D. Gabbay and U. Reyle, N-Prolog: An Extension to Prolog with Hypothetical Implications I. Journal of Logic Programming 1(4), 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. Gordon, R. Milner, and C. Wadsworth, *Edinburgh LCF*, Springer Verlag, 1979.

[6] L. Hallnäs and P. Schroeder-Heister, A Proof-Theoretic Approach to Logic Programming I. Generalized Horn Clauses. SICS research report R88005.

[7] J. Hannan and D. Miller, Enriching a Meta-Language with Higher-Order Features. Workshop on Meta-Programming in Logic Programming, Bristol, June 1988.

[8] J. Hannan and D. Miller, Uses of Higher-Order Unification for Implementing Program Transformers. Fifth International Conference Symposium on Logic

Programming, MIT Press, 1988.

[9] R. Harrop, Concerning Formulas of the types $A \rightarrow B \vee C,\ A \rightarrow (Ex)B(x)$ in Intuitionistic Formal Systems. Journal of Symbolic Logic, 25(1), 1960, 27 — 32.

[10] G. Huet, A Unification Algorithm for Typed λ-Calculus. Theoretical Computer Science 1, 1975, 27 – 57.

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

[12] J. Jaffar and J.-L. Lassez. Constraint Logic Programming. Fourteenth Annual ACM Symposium on Principles of Programming Languages, Munich, January 1987, 111–119.

[13] J. Lloyd and R. Topor, Making Prolog More Expressive. Journal of Logic Programming 1(3), October 1984, 225 – 240.

[14] L. McCarty, Clausal Intuitionistic Logic I. Fixed Point Semantics. Journal of Logic Programming 5(1), March 1988, 1 – 31.

[15] L. McCarty, Clausal Intuitionistic Logic II. Tableau Proof Procedure. To appear in the Journal of Logic Programming.

[16] D. Miller, A Theory of Modules for Logic Programming. Symposium on Logic Programming, Salt Lake City, 1986, 106 – 115.

[17] D. Miller, Hereditary Harrop Formula and Logic Programming. VIII International Congress of Logic, Methodology, and Philosophy of Science, Moscow, August 1987.

[18] D. Miller, A Logical Analysis of Modules in Logic Programming. To appear in the Journal of Logic Programming.

[19] D. Miller, Unification Under a Mixed Prefix. Unpublished, June 1988.

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

[21] D. Miller and G. Nadathur, Higher-Order Logic Programming. Third International Logic Programming Conference, London, June 1986, 448 – 462.

[22] D. Miller and G. Nadathur, A Logic Programming Approach to Manipulat-

ing Formulas and Programs. IEEE Symposium on Logic Programming, San Franciso, September 1987.

[23] D. Miller, G. Nadathur and A. Scedrov, Hereditary Harrop Formulas and Uniform Proofs Systems. Second Annual Symposium on Logic in Computer Science, Cornell University, June 1987, 98 — 105.

[24] D. Miller, G. Nadathur, F. Pfenning, and A. Scedrov, Uniform Proofs as a Foundation for Logic Programming. Submitted.

[25] R. Milner, A Theory of Type Polymorphism in Programming. Journal of Computer and System Sciences 17 (1978) 348 – 375.

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

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

[28] L. Paulson, The Foundations of a Generic Theorem Prover. University of Cambridge technical report 130, March 1988.

[29] F. Pfenning, Partial Polymorphic Type Inference and Higher-Order Unification. ACM Conference on Lisp and Functional Programming, Snowbird, Utah, July 1988.

[30] F. Pfenning and C. Elliot. Higher-Order Abstract Syntax. ACM-SIGPLAN Conference on Programming Language Design and Implementation, 1988.

[31] D. Prawitz, *Natural Deduction,* Almqvist & Wiksell, Uppsala, 1965.

[32] L. Sterling and E. Shapiro, *The Art of Prolog: Advanced Programming Techniques*, MIT Press, Cambridge MA, 1986.