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 \( \lambda \)-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, \( \lambda \)Prolog 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 \( \lambda \)-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 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 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 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 S of sorts or primitive types, a set \( \Phi \) of type variables, and a set C of type constructors where each type constructor has a unique positive arity. The types of T are then defined inductively by the following rules: