CS-1994-35

Scoping Constructs in Logic Programming: Implementation Problems and their Solution

Gopalan Nadathur
Bharat Jayaraman
Keehang Kwon

Department of Computer Science

Duke University

Durham, North Carolina 27708–0129

October 1994

Scoping Constructs in Logic Programming: Implementation Problems and their Solution†

Gopalan Nadathur ‡
Department of Computer Science
Duke University, Durham, NC 27706
gopalan@cs.duke.edu

Bharat Jayaraman
Department of Computer Science
State University of New York at Buffalo
Buffalo, NY 14260
bharat@cs.buffalo.edu

Keehang Kwon
Department of Computer Science
Duke University, Durham, NC 27706
kwon@cs.duke.edu

Abstract

The inclusion of universal quantification and a form of implication in goals in logic programming is considered. These additions provide a logical basis for scoping but they also raise new implementation problems. When universal and existential quantifiers are permitted to appear in mixed order in goals, the devices of logic variables and unification that are employed in solving existential goals must be modified to ensure that constraints arising out of the order of quantification are respected. Suitable modifications that are based on attaching numerical tags to constants and variables and on using these tags in unification are described. The resulting devices are amenable to an efficient implementation and can, in fact, be assimilated easily into the usual machinery of the Warren Abstract Machine (WAM). The provision of implications in goals results in the possibility of program clauses being added to the program for the purpose of solving specific subgoals. A naive scheme based on asserting and retracting program clauses does not suffice for implementing such additions for two reasons. First, it is necessary to also support the resurrection of an earlier existing program in the face of backtracking. Second, the possibility for implication goals to be surrounded by quantifiers requires a consideration of the parameterization of program clauses by bindings for their free variables. Devices for supporting these additional requirements are described as also is the integration of these devices into the WAM. Further extensions to the machine are outlined for handling higher-order additions to the language. The ideas presented here are relevant to the implementation of the higher-order logic programming language λProlog.

† This paper has been accepted for publication in the *Journal of Logic Programming*.

‡ This address is functional only until January 1, 1995. After this date, please use the following address: Department of Computer Science, University of Chicago, Ryerson Laboratory, 1100 E. 58th Street, Chicago, IL 60637, Email: `gopalan@cs.uchicago.edu`.

1 INTRODUCTION

This paper examines techniques relevant to the implementation of the logic programming language λProlog [21]. The basis for this language is provided by a polymorphic version of the logic of higher-order hereditary Harrop or *hohh* formulas [17]. At a qualitative level, the logic of *hohh* formulas represents an amalgamation of extensions to Horn clause logic in two different directions. The extension in one direction is obtained by including higher-order features — in the form of quantification over function and some occurrences of predicate variables and the replacement of first-order terms by simply typed lambda terms — within Horn clauses, thereby producing the logic of *higher-order* Horn clauses [22]. Along the other direction, Horn clause logic is enhanced by permitting universal quantifiers and restricted uses of implications, resulting in a first-order version of the logic of hereditary Harrop formulas [13, 17]. The combination of these two logics produces a simply typed version of the logic of *hohh* formulas. The typing paradigm incorporated in this logic is somewhat constraining from the perspective of programming. However, it can be relaxed through the introduction of polymorphism. The resulting logic is what constitutes the basis for λProlog.

The enrichments to Horn clause logic that are embodied in the logic underlying λProlog provide for new features at a programming level. λProlog is, in fact, a language that manifests these features and consequently has several novel capabilities in comparison with a language like Prolog. The usefulness of these capabilities has lead to a significant interest in the language and systems have been developed that implement λProlog or a close relative of it [2, 4, 21]. These systems notwithstanding, there has been little discussion of techniques that are well-suited to the implementation of such a language.[1] The considerations in this paper are part of an effort that focuses on precisely this issue, with the ultimate goal of providing an efficient and robust implementation for λProlog. We have found the hierarchy of logics described above a useful structuring device in this endeavor. In particular, we have been developing an implementation scheme for the full language by starting with the Warren Abstract Machine (WAM) [26], which is usually employed for Prolog, and considering independently the new devices that are required for dealing with higher-order aspects, types, and implications and universal quantifiers. There is good reason for adopting such an approach: Unification and backtracking are central to the implementation of all the logics in question, and the WAM provides a good framework for an efficient treatment of these aspects. Furthermore, the new features in λProlog are in a sense orthogonal to each other. Consequently there is little interference between the mechanisms developed for realizing each of these features, and, in fact, they blend together well in an overall machine.

In keeping with the above strategy, this paper discusses implementation methods for one of the new aspects of λProlog, namely, the provision of implications and universal quantifiers in goals. It complements, in this respect, other work that we have done concerning the treatment of higher-order aspects [19, 20, 23] and types [11]. The particular enrichment considered here is also of interest in its own right: permitting implications and universal quantifiers in goals provides the basis for scoping constructs in a language such as Prolog. From the perspective of an implementation, the inclusion of these symbols gives rise to two new kinds of problems. The first kind of problem arises from the possibility of alternating sequences of universal and existential quantifiers appearing in goals. Solving a universally quantified goal requires the introduction of a "new" constant. The usual

[1]There is, however, a discussion of the implementation problems in [4] and also a systematic development of an interpreter for λProlog within a functional programming language.

implementation technique employed for an existential quantifier is to instantiate it with a "logic" variable whose value is determined at a later stage through unification. Care must be exercised in combining these two strategies. In order to guarantee the newness of a constant introduced for a universal quantifier, this constant must not be allowed to appear in the term that ultimately instantiates a surrounding existential quantifier. A proper treatment of unification is necessary for this purpose. The second kind of problem is caused by the fact that implications in goals require sets of program clauses to be periodically added and removed from the program. While it might seem that a simple-minded stack-based scheme can be used to implement programs that change in this manner, there are some complications: First, the program clauses that may need to be assumed may be "parameterized" by bindings for the free variables occurring in them, requiring them to be treated as *closures*. Second, backtracking action may require the reinstatement of a program in existence at some earlier point and a bookkeeping scheme that makes it possible to carry out this action in an efficient manner is needed.

In the rest of this paper, we discuss in detail the provision of implications and universal quantifiers in goals and the new implementation problems that arise from this enhancement. This discussion is structured as follows. In the next section we present informally a language that extends Prolog in the manner mentioned and we illustrate the usefulness of the new features of this language. We describe a first-order version of this language formally in Section 3 and discuss the implementation problems that arise in its context. We then devote our attention to methods for dealing with these problems. In Section 4 we present an abstract interpreter for our extended language that contains within it a conceptual scheme for handling universal quantifiers. This interpreter is naive in its treatment of implication goals, and the next two sections focus on this issue. In Section 5 we present solutions to the two main problems that arise in this context: the parameterization of program clauses and the need to resurrect old program contexts on backtracking. An efficient realization of these solutions within a WAM-like framework is discussed in Section 6. In Section 7, we examine the possibility of compilation within our implementation scheme. This discussion provides a complete picture of our implementation ideas and also illustrates the graceful manner in which the additional machinery fits into that of the WAM. Although the most interesting motivating examples for the inclusion of implications and universal quantifiers in goals involve the use of a higher-order language, simplicity of exposition dictates that we present our implementation ideas in a first-order context. We amend this situation in Section 8 by indicating how these ideas translate to the higher-order context and by describing methods for dealing with additional aspects of scoping in this context that are not covered by them. We conclude this paper in Section 9.

2 USES OF IMPLICATIONS AND UNIVERSAL QUANTIFIERS IN GOALS

We shall describe precisely the idea of a "goal" in Section 3; for the moment, this may be understood as what can appear as the body of a Prolog clause or what can be written as a user query. From a logical perspective, the syntax for goals in Prolog is the following: they can be atomic formulas or conjunctions or disjunctions of simpler goals. In the context of Prolog, conjunctions are written using commas and disjunctions are written using semicolons. Although no explicit syntax is provided for this purpose, existential quantification may also be present in goals. Thus, a clause of the form $\forall x\,(B(x) \supset H)$, written as `H :- B(X)` in Prolog, is equivalent (in classical logic) to

$(\exists x\, B(x) \supset H)$ if x does not appear in H.

The language whose implementation we wish to consider is one in which this set of logical symbols is extended to include implications and universal quantifiers. In this language, formulas such as $F \supset G$ and $\forall x\, G$ will be permitted as goals, provided G is itself a goal. The intended semantics of these two new operations is the following. A goal of the form $F \supset G$ is to be solved by adding F to the current program and then solving G. This places a constraint on F: it should have the structure of a conjunction of program clauses. Given this understanding, implications provide a device for giving program clauses a scope. Thus, F is to be available only in the course of solving G. As for a goal of the form $\forall x\, G$, it is intended to be solved by instantiating x in G with a new constant c and then solving the resulting goal. Interpreted in this fashion, the universal quantifier provides a means for limiting the availability of names. The universally quantified variable is, in fact, a name that is visible only within the scope of the quantifier.

Based on the informal understanding of the new symbols, it is not difficult to imagine that their addition to Prolog might be valuable. A problem with Prolog is that there is no structure to its program and name spaces. A program is a monolithic piece of code and all the predicates defined and constants used in one place are visible everywhere else. It is well appreciated that this is an undesirable characteristic for a programming language. Implications and universal quantifiers provide a means for introducing some structure.

2.1 Lexical Scoping

An example illustrating the problem mentioned above is provided by auxiliary definitions. Thus, consider the definition of the reverse relation for lists in Prolog. A naive definition of this relation is provided by the clauses[2]

```
rev([],[]).
rev([X|L1],L2) :- rev(L1,L3), append(L3,[X],L2).
```

where `append` is defined by the usual set of clauses for appending lists. As is well known, this definition of the reverse relation is inefficient. The execution time of the `append` program is linear in the length of the list that is its first argument. Its repeated invocation in the course of reversing a list results in a program that takes time that is quadratic in the length of the input list.

A more efficient reverse program can be written by using the idea of an accumulator. The following definition of `rev` embodies this idea:

```
rev(L1,L2) :- rev_aux(L1,L2,[]).

rev_aux([],L,L).
rev_aux([X|L1],L2,L3) :- rev_aux(L1,L2,[X|L3]).
```

The declarative interpretation of `rev_aux` here is that it is true of three lists if the second is the result of appending the reverse of the first to the last. The point to note with this example is that `rev_aux` is an extremely specialized predicate whose only purpose for existence is its usefulness in defining `rev`. However, its definition in Prolog occurs at the same level as that of `rev`. There are at least two undesirable consequences of this. First, a scan of the program does not suffice for

[2]In the examples in this section, we use standard Prolog syntax, mixed with the obvious syntax for implications and universal quantifiers. The reader unfamiliar with Prolog syntax is referred to a Prolog text such as [3].

determining what role is played by each predicate defined in it. Second, it is possible for the name rev_aux to be confused with the name of some other relation that is defined at this level, thereby producing a mixture of definitions.

Permitting implications in goals provides a means for solving some of these problems. The definition of rev_aux can be made "local" to that of rev as indicated below:

```
rev(L1,L2) :-
(((∀L rev_aux([],L,L)) ∧
  (∀X ∀L1 ∀L2 ∀L3 (rev_aux([X|L1],L2,L3) :- rev_aux(L1,L2,[X|L3]))))
          ⊃ rev_aux(L1,L2,[])).
```

As an explanation of the syntax of this clause, it is obtained by moving the two clauses defining rev_aux into the body of the clause defining rev. When appearing in the body of a clause, a set of clauses must be represented by a conjunction and the quantification of variables in these clauses (that was earlier left implicit) must now be made explicit. Factoring this in should make the structure of the clause above clear. The following points might be observed with regard to this modified definition of rev. First, the clauses defining rev_aux are not available at the top-level. Thus, these clauses will not affect the meaning of this predicate if it is defined through some other clauses at that level. Second, an additional structure is added to the program that helps in understanding the purpose of its parts. For example, it is clear merely from looking at the clause above that rev_aux must be an auxiliary definition for rev. Finally, while the definition of rev_aux is not available at the top-level, the semantics described for implication ensures that it will become available in the course of solving the body of rev. Thus, consider the invocation of the goal rev([1,2,3],L). This results in the goal rev_aux([1,2,3],L,[]) being invoked after the program has been dynamically augmented with the formula

```
(∀L rev_aux([],L,L)) ∧
(∀X ∀L1 ∀L2 ∀L3 (rev_aux([X|L1],L2,L3) :- rev_aux(L1,L2,[X|L3])))
```

The universal quantifiers and conjunction can now be made implicit, revealing that the desired definition of rev_aux is indeed available at this stage.

The use of an implication in the body of a program clause thus provides the effect of block structuring. This gives meaning to the notions of global and local variables within logic programming. As an example, using a global variable, we can eliminate the "result" argument of rev_aux in the definition of rev, and use the following definition instead:

```
rev(L1,L2) :-
    ((rev_aux([],L2) ∧
      (∀X ∀L1 ∀L3 (rev_aux([X|L1],L3) :- rev_aux(L1,[X|L3]))))
              ⊃ rev_aux(L1,[])).
```

Notice that the variable L2 is "shared" between rev and rev_aux. As can be seen from tracing the computation involved in a query such as rev([1,2,3],L), this variable eventually provides a means for communicating the result out to the top-level. Communication in the other direction — a standard fare in a functional programming language such as ML — can also occur and has its uses. In either case, we note that the execution of a query may require the addition of a special kind of clause, in particular, a clause with "tied" variables, to the program. For example, the query rev([1,2,3],L) would result in the clauses

```
rev_aux([],L2).
rev_aux([X|L1],L3) :- rev_aux(L1,[X|L3]).
```

being added to the program. Following the earlier suggestion, we have dropped the quantifiers and the conjunction. Note, however, that the variable `L2` that appears in the first clause is *not* universally quantified over the clause. Rather, it has a binding determined dynamically at the point that it is added to the program and is in fact identical to the variable `L` in the query.

While the use of an implication goal helps solve some of the problems mentioned in connection with the initial definition of `rev`, one problem still remains. The meaning of the predicate `rev_aux` inside the body of `rev` is not insulated from definitions in existence outside the body. Thus, if the global program contains other clauses defining `rev_aux`, the invocation of the implication goal does not cause a replacement of these by two new clauses but, rather, only an addition of the two clauses to the existing collection. This might be the desired effect in certain situations but clearly not in the present one.

The problem under consideration can be viewed as one of limiting the scope of the name `rev_aux` and can be solved as such by using a universal quantifier. In particular, the definition of `rev` can be rewritten as follows:

```
rev(L1,L2) :-
(∀rev_aux ((rev_aux([],L2) ∧
           (∀X ∀L1 ∀L3 (rev_aux([X|L1],L3) :- rev_aux(L1,[X|L3]))))
              ⊃ rev_aux(L1,[]))).
```

To understand this definition, let us suppose that the goal `rev([1,2,3],L)` is invoked. This leads to an attempt to solve the goal

```
∀rev_aux ((rev_aux([],L2) ∧
          (∀X ∀L1 ∀L3 (rev_aux([X|L1],L3) :- rev_aux(L1,[X|L3]))))
             ⊃ rev_aux(L1,[])).
```

The indicated semantics of the universal quantifier dictates picking a new name for `rev_aux` and then solving the instantiation of the given query with this name. Once a name is picked, the remainder of the computation proceeds as before. However, the fact that a new name is chosen for `rev_aux` ensures the desired insulation from the effects of outside definitions.

2.2 Data Abstraction

The universal quantifier is used in the above example to hide the name of a predicate. In a similar fashion, it may be used to hide the names of function and constant symbols. These symbols serve to determine the representation of data in logic programming. The ability to hide their names therefore has the potential of supporting data abstraction.

To illustrate this possibility, let us assume that we wish to develop a program that uses a store. A program of this sort may be one that carries out a graph search. Now, the development of this program can be divided into two conceptually different tasks: (a) the implementation of graph search using an abstract model of the store and (b) the implementation of the store. From the perspective of the first task, we may look upon a store as being given by three operations: $empty(S)$ that initializes S to the empty store, $remove(X, S_1, S_2)$ that produces the store S_2 by removing the item X from S_1 and $add(X, S_1, S_2)$ that produces the store S_2 by adding item X to S_1. An implementation of graph search can now be provided that makes no assumptions concerning the actual implementation of these operations.

While this kind of data abstraction might be used at a conceptual level, no language-level support is provided for it in Prolog. For example, let us suppose that the store is represented by a stack and the operations mentioned above are implemented through the following clauses:

```
empty(emp).
remove(X,stk(X,S),S).
add(X,S,stk(X,S)).
```

Despite the programmer's best intentions, the actual representation of the stack, embodied in the symbols `emp` and `stk`, is visible everywhere in the program and may be freely used in the procedures that implement graph search. We also observe that the predicates implementing the operations on the store are visible at the top-level instead of being available only within the graph search procedures.

Universal quantifiers and implications can be used to alleviate both the problems mentioned above. Thus, let us assume that the store is in fact needed for implementing graph search and that interface to the graph search procedures is provided through a predicate of one argument called `graph_search`. Then the definition of the store may be relativized to the invocation of `graph_search` by using the following query:

```
∀emp ∀stk ((empty(emp) ∧
            (∀X ∀S remove(X,stk(X,S),S)) ∧
            (∀X ∀S add(X,S,stk(X,S))))
               ⊃ graph_search(Solution))
```

Solving this query requires introducing new names for the quantified variables `emp` and `stk`, thereby ensuring that the "names" `emp` and `stk` that are used within the implementation of the store are not confused with names appearing anywhere else in the program. The semantics of implication ensures that the operations on the store are defined at the time the procedure `graph_search` is invoked, and hence they may be freely used within this procedure and its auxiliary procedures. Notice that these procedures cannot inspect the representation of the store; in particular, they cannot access (the new constants that replace) `emp` and `stk` directly. However, they can still use the store and can communicate through "store valued" variables. Note also that there is a sense of modularity to the code presented. The procedures implementing the store operations can be replaced by a different implementation without affecting the usability of the graph search procedures.

The various ideas described here show that implications and universal quantifiers can be used to realize notions of modules and abstract datatypes in logic programming. A fuller development of these ideas can be found in [14] and [15].

2.3 Metalanguage Aspects

Prolog has certain features that make it a natural choice for prototyping reasoning systems: it supports the idea of search in an intrinsic way and its embodiment of first-order terms and unification leads to convenient ways for representing and manipulating the objects that are to be reasoned about. However, there are ways in which its abilities in this direction can be improved. For instance, it has been argued (*e.g.*, see [16, 25]) that using lambda terms instead of first-order terms provides for an even better representation of the objects that are to be manipulated. More relevant to the present paper is the addition of the search primitives contained in the new logical symbols being considered. One scenario that occurs frequently in reasoning tasks is that of making an assumption and then trying to reach a conclusion. This kind of *hypothetical* reasoning is supported very naturally by implication, given our interpretation of this symbol. Another paradigm that is useful is that of introducing a new object and then determining if a given statement is true of it. This is the basis, for instance, of universal generalization. Universal quantifiers in goals provide a means for realizing this paradigm.

We illustrate the above observations by considering the task of type inference for lambda terms.[3] These terms are constructed from constants and variables using the operations of abstraction and application. We assume that the types of constants are previously specified. The types of variables are determined by an *environment*. An arbitrary lambda term can then be inferred to be of a certain type relative to an environment Γ by using the following rules:

(i) An occurrence of a constant has as a type any instance of the type specified for the constant.

(ii) Every occurrence of a variable has as its (sole) type the one assigned to the variable by Γ.

(iii) If t_1 and t_2 have $\alpha \rightarrow \beta$ and α as types relative to Γ, then $(t_1\ t_2)$ has β as a type relative to Γ.

(iv) If t has β as a type relative to an environment that is like Γ except that that it assigns the type α to x, then $\lambda x\, t$ has $\alpha \rightarrow \beta$ as a type relative to Γ.

Types are assumed to be polymorphic here, and are represented by first-order expressions with the single binary infix function symbol $\rightarrow$ and a collection of constant symbols that represent the primitive types.

Suppose now that we wish to write a logic program that infers types for closed lambda terms. This program will need, first of all, to associate types with constants. These associations can be represented through facts or atomic clauses. The program will also need to represent an environment. Since our interest is in inferring types for closed terms, it is necessary only to maintain the types assigned to bound variables by the environment. Thus, the environment may also be represented by means of a set of facts, with implication goals being used to add to this set at the point where abstractions are encountered. To provide concreteness to this discussion, let us suppose that the only constants available are 1 and + of type *int* and $int \rightarrow (int \rightarrow int)$ respectively. Then the following program represents an attempt at implementing type inference using these ideas:

```
type_of(1,int).
type_of(+, int → (int → int)).
type_of(app(E1,E2),T1) :- type_of(E1,T2 → T1), type_of(E2,T2).
type_of(abst(X,E),T1 → T2) :- (type_of(X,T1) ⊃ type_of(E,T2)).
```

[3] We assume a familiarity in the rest of this section with basic lambda calculus notions. The reader unfamiliar with these may consult [8] or some similar source.

We have assumed a first-order representation of lambda terms in this program, with abstraction and application being represented by the binary function symbols `abst` and `app` respectively, and (object-language) constants by suitably chosen constant symbols.

A question that is not yet settled in connection with the above program is whether variables in lambda terms are to be represented by variables or constants of the programming language. A brief consideration of this question leads to the conclusion that (metalanguage) variables are not the right choice: using such a representation would permit, for instance, the erroneous inference that $\lambda x\, \lambda y\, ((+\ x)\ y)$ has $\alpha \rightarrow (\beta \rightarrow int)$ as one of its types for any choice of types for α and β. Unfortunately, there is a problem with the program shown even if constants are used to represent the variables in lambda terms. The source of this problem is that the same variable name may be used for different abstractions in a given lambda term and, in this case, an inner abstraction is intended to "hide" the outer abstraction. It is by virtue of this convention that a term such as $\lambda v\, \lambda v\, ((+\ v)\ (v\ v))$ is deemed to be ill-typed. However, this hiding effect is not realized by our program. Thus, assuming that lambda term variables are represented by constants of the same name, the term $\lambda v\, \lambda v\, ((+\ v)\ (v\ v))$ will be judged by our program to have $(int \rightarrow int) \rightarrow (int \rightarrow int)$ as one of its types.

The problem can be solved by using universal quantifiers. However, we need to change our representation of lambda terms before we can describe this solution. To begin with, we assume that the data structures of our language are themselves provided by lambda terms and not first-order terms. We do this because we need an encoding of substitution in the solution we provide, and using lambda terms as data structures leads to this being available as a primitive operation. Now, we represent an object-language expression such as $\lambda x\, E$ by `abst(λx E)` where `E` is the translation of E (with x replaced by `x`). The scheme for representing applications remains unchanged. Using this representation of lambda terms, a correct type inference program is given by the following clauses:

```
type_of(1,int).
type_of(+, int → (int → int)).
type_of(app(E1,E2),T1) :-
          type_of(E1,T2 → T1), type_of(E2,T2).
type_of(abst(E),T1 → T2) :-
          (∀x (type_of(x,T1) ⊃ type_of(E(x),T2))).
```

The manner in which the universal quantifier in the body of the third clause serves to introduce a new constant dynamically should be noted in this example. This constant must be substituted into the body of the abstraction. By virtue of our representation of terms, this effect is produced by the application of `E` to the quantified variable `x`. This application is written in the program above as `E(x)`.

The examples considered in this section are simple ones, intended only to bring out the semantics of the new logical symbols and the value of their inclusion in logic programming. More extensive examples may be found in various places in the literature. (See, for example, [5, 7, 14, 15, 24].) In the following sections we provide a precise definition of a logical language that includes implications and universal quantifiers in goals and we examine the implementation of this language. The language that we consider is a first-order one and does not explicitly cover all the examples presented here. This simplification is chosen largely for expository reasons and nothing essential to the implementation of implications and universal quantifiers in goals is left out by it. Towards bringing this point out, we indicate in Section 8 the additional devices necessary for handling the higher-order aspects present in the examples of this section.

3 AN EXTENDED LANGUAGE AND THE PROBLEMS IN ITS IMPLEMENTATION

A language that utilizes implications and universal quantifiers can be described as an extension to the language based on Horn clauses. In describing this extension, we need to adopt a somewhat more general view of Horn clauses than is usual. Based on the methodology developed in [17], the logic underlying a logic programming language may be characterized by two classes of formulas: the G formulas that function as goals or queries, and the D formulas that fill the role of program or, to use a terminology common in discussions of Horn clauses, definite clauses. In this context, the programming framework provided by Horn clauses is defined by the G and D formulas given by the following syntax rules in which A represents an atomic formula:

$$
\begin{aligned}
G &::= A \mid (G \wedge G) \mid (G \vee G) \mid (\exists x\, G),\\
D &::= A \mid (G \supset A) \mid (\forall x\, D).
\end{aligned}
$$

The parentheses that surround expressions in these and other syntax rules are included to ensure unique readability and may be omitted if doing so does not cause an ambiguity. Now, the formulas described above are related to Horn clauses in the following sense: within the setting of classical logic, the negation of a G formula is equivalent to a set of negative Horn clauses and, similarly, a D formula is equivalent to a set of positive Horn clauses. The syntax adopted here is motivated by its greater proximity to actual programming realizations, its amenability to extensions and our use of derivability, as opposed to refutability, as the primitive semantic notion.

In the framework of [17], the task of programming consists of describing a set of relationships between objects through a collection of closed program clauses thought of as a program, and of querying such a specification through goals. From a logical perspective, this viewpoint is justified only if the task of answering a query can be equated with the notion of constructing a proof for the query from the given program. In the context of Horn clauses, use can be made of either classical or intuitionistic provability to satisfy this requirement. Both derivability relations validate the following recipe for solving a closed goal G given a program $\mathcal{P}$:

(1) if G is $G_1 \wedge G_2$ then try to solve it by solving both G_1 and G_2,

(2) if G is $G_1 \vee G_2$, then try to solve it by solving either G_1 or G_2,

(3) if G is $\exists x\, G_1$, then try to solve it by solving $[t/x]G_1$ for some closed term t, and

(4) if G is an atom, then try to solve it (a) by determining that it is an instance of a program clause in $\mathcal{P}$, or (b) by finding an instance $G_1 \supset G$ of a program clause in $\mathcal{P}$ and trying to solve G_1.

The program is assumed to be fixed throughout the above description, and the notation $[t/x]G$ is used to denote the result of replacing every free occurrence of x in G by t, taking care, of course, to avoid the inadvertent capture of free variables. The most interesting aspect of the above recipe is that it permits the connectives and quantifiers in goals to be interpreted dually as search primitives. Under this interpretation, $\vee$ and $\wedge$ respectively specify OR and AND branches in a search and the existential quantifier specifies an infinite OR branch with the branches parameterized by closed terms. The behavior of existential quantifiers also permits "answers" to be extracted from computations: a goal with free variables may be interpreted as a request to solve the existential closure of the formula and to produce instantiations for the introduced quantifiers that lead to successful solutions.

The extended language that we desire must permit implications and universal quantifiers in goals. These additions are incorporated into a language that is based on first-order hereditary Harrop or *fohh* formulas [17]. The syntax of goals and program clauses in such a language is given by the G and D formulas described by the following rules:

$$
\begin{aligned}
&G ::= A \mid (G \wedge G) \mid (G \vee G) \mid (\exists x\, G) \mid (Ds \supset G) \mid (\forall x\, G),\\
&Ds ::= D \mid (D \wedge Ds), \text{ and}\\
&D ::= A \mid (G \supset A) \mid (\forall x\, D).
\end{aligned}
$$

Note that the implications that are permitted in goals in this extended language are limited — conjunctions of D formulas must appear on the left and G formulas on the right. However, this restriction is in keeping with our informal discussion in the previous section.

Our desire is to interpret implication and universal quantification as scoping mechanisms with respect to program clauses and names respectively. This is exactly the effect we obtain if the idea of solving a goal with respect to a program is clarified using the notion of intuitionistic provability. In particular, if $\mathcal{P}$ is the program, then the following additions need to be made to the earlier recipe to get one for solving a closed goal G in the new language:

(5) if G is $(D_1 \wedge \ldots \wedge D_n) \supset G_1$, then try to solve it by solving G_1 using $\mathcal{P} \cup \{D_1, \ldots, D_n\}$ as the program instead of $\mathcal{P}$, and

(6) if G is $\forall x\, G_1$, then try to solve it by solving $[c/x]G_1$ for some new constant c.

The recipes for solving a goal from a program that are provided above are useful in understanding the nature of computation in the languages that are based on Horn clauses and on *fohh* formulas. They also provide some indication of how computations might actually be carried out. However, they are not complete from this perspective. One problem is that the instruction for solving existential goals assumes an oracle for picking the "right" instantiation for the quantifier. Similarly, choices have to be made concerning the disjunct to be solved in a disjunctive goal and the program clause to be used in solving an atomic goal. In each of these cases, some machinery is needed in addition to the basic instruction to support the making of these choices.

The additional machinery that suffices for implementing the Horn clause language is, by now, quite standard. The problem with existential quantifiers is dealt with by delaying the actual instantiations of such quantifiers till such time that information is available for making an appropriate choice. This effect is achieved by replacing the quantified variables by placeholders whose values are determined later through the process of unification. Thus, a goal such as $\exists x\, G(x)$ is transformed into one of the form $G(X)$ where X is a new logic variable that may be instantiated at a later stage. In attempting to solve an atomic goal A, we look for a definite clause $\forall \bar{y}\,(G' \supset A')$ such that A unifies with the atomic formula that results from A' by replacing the universally quantified variables with new logic variables. If such a clause is found, the next task becomes that of solving the resulting instance of G'. The approach that is used to deal with the other forms of nondeterminism is to assume an implicit ordering of choices and to implement a depth-first search with the possibility of backtracking; thus, disjunctive goals are considered in left-to-right order and program clauses are used in the order of presentation. A final point to note is that much of the unification, the processing of the search primitives, and the sequencing through program clauses can be compiled within this framework. These various observations are in fact used in WAM-based approaches to provide extremely efficient implementations for the programming paradigm based on Horn clauses.

Our desire in this paper is to extend these methods to obtain a satisfactory implementation of a programming language based on *fohh* formulas. It may appear that such an extension can be easily obtained: in order to deal with universal quantifiers, we merely need to consider the instantiation of a goal with a newly generated constant and to deal with implications we only need a mechanism for adding program clauses to a program. However, an implementation based solely on this view would be both incorrect and inadequate.

The suggestion for dealing with universal quantifiers ignores interactions with the scheme being built upon and would be erroneous if executed naively. The source of the problem is that universal and existential quantifiers can appear in arbitrary orders in the goals that are of interest. For example, consider the task of solving the goal $\exists x\, \forall y\, p(x, y)$, where p is a predicate symbol. Using the scheme suggested, we may reduce this task to that of solving the "goal" $p(X, c)$ where c is a new constant and X is a logic variable. However, unification cannot be used in an unqualified fashion in solving the new goal because any instantiation that is determined for X must not contain c in it. Thus, suppose that we attempt to solve the goal $\exists x\, \forall y\, p(x, y)$ in the context of a program containing the clause $\forall x\, p(x, x)$. If care is not exercised, an incorrect derivation for the goal may be constructed by (indirectly) instantiating X to c.[4]

In providing a satisfactory treatment of implications in goals, there are several aspects that require a detailed consideration. First, we observe that in its presence the program being used cannot be left implicit. Thus, consider solving the goal $(D_1 \supset G_1) \wedge (D_2 \supset G_2)$ from a program $\mathcal{P}$. This task eventually requires two different programs, *i.e.*, $\mathcal{P} \cup \{D_1\}$ and $\mathcal{P} \cup \{D_2\}$, to be used in solving the goals G_1 and G_2. An acceptable implementation should not require the explicit construction of two separate programs but rather should support the realization of the two different contexts through a process of gradual addition and removal of code. Such a scheme can actually be supported and the implementation we describe later even permits the compilation of program clauses that are to be added to the original program. However, the interaction of backtracking with this approach requires bookkeeping devices of some sophistication. To see why this is the case, consider solving the goal $\exists x\, ((D \supset G_1(x)) \wedge G_2(x))$ from the program $\mathcal{P}$. Under the scheme being considered, we would first have to augment the program with D and attempt to solve the goal $G_1(X)$, where X is the logic variable introduced for the existential quantifier. A successful solution would determine a binding for X. An attempt would now have to be made to solve the appropriate instance of $G_2(X)$ after removing D from the program. Assume now that, with the

[4]In the context of classical logic, universal quantifiers can be eliminated by using Skolem functions of the existentially quantified variables within whose quantifier scope they appear. (Note that the roles of quantifiers are reversed in the refutability setting.) Incorrect instantiations of the kind discussed above will then be blocked by the process of "occurs-checking" in unification. Unfortunately, as discussed in [18], the problem cannot be dealt with in the present context in a similar "static" fashion. Some feeling for this might be obtained by trying to determine how the static process ought to work in conjunction with the goal $(\forall x\, p(x) \supset q) \supset \exists x\, (p(x) \supset q)$, noting that this goal should not succeed. However, a dynamic form of Skolemization does work even in this context. The solution used in this paper captures the constraints dynamic Skolemization is designed to capture in a much more direct fashion.

instantiation found for X, $G_2(X)$ cannot be solved. The requirement then is to look for another solution to $G_1(X)$. However, such a solution must be sought in the context of the relevant program; in particular, the program clause D must be reinstated and any additions made in the course of trying to solve $G_2(X)$ must be removed. In general, we see that backtracking may require the program that is to be used to be changed substantially, and mechanisms have to be provided for realizing such switches in context in an efficient manner.

The final problem concerns the presence of tied variables in program clauses. One situation in which this arises is that when existential quantifiers are used in conjunction with implications. Thus, consider solving the goal $\exists x\,(p(x) \supset g(x))$ where p and g are predicate names. Assuming that x is replaced by the logic variable X and implication is dealt with in the manner required, we would have to solve the goal $g(X)$ with respect to a program that contains the clause $p(X)$. Notice that the variable that occurs in $p(X)$ is different from the variables that usually occur in program clauses: it cannot be instantiated in arbitrary fashion but rather only in one particular way that is also consistent with the instantiation for the occurrence of the same variable in the goal $g(X)$. To appreciate this aspect completely, consider solving the given goal from a program containing the clauses $q(a)$ and $\forall x\,((q(x) \wedge p(b)) \supset g(x))$, assuming that a and b are constants and q is a predicate name. It may appear at first that the goal should succeed in this context. Thus, we may backchain on the second clause in the original program to solve $g(X)$, producing the subgoals $q(X)$ and $p(b)$. The subgoal $q(X)$ might be solved by using the clause $q(a)$, and the subgoal $p(b)$ may apparently be solved by using the program clause $p(X)$. Such a solution would in reality be erroneous: the variable in the program clause $p(X)$ is tied to the one in the goal $g(X)$ and, thus, this "solution" involves instantiating the logic variable X simultaneously with a and b. More generally, we see that a suitable implementation of our language must contain mechanisms for distinguishing between variables of two different kinds that might now appear in programs and also for dealing with the new kind of variables.

In the remainder of this paper we develop methods for dealing with the various new implementation problems that arise in the context of a language that is based on *fohh* formulas. We shall describe these methods as extensions to the machinery already present in the WAM. Two questions need to be answered in justifying this approach: why is the WAM used as a starting point and might not a metaprogramming approach, perhaps using the impure predicates present in Prolog, yield a satisfactory result as well? We have discussed in this section the manner in which a language that is based on *fohh* formulas builds on one that is based on Horn clauses and have also motivated the use of several mechanisms present in implementations of the latter in obtaining an implementation of the former. This discussion provides a strong argument for utilizing the structure of the WAM in implementing the language that is presently of interest. Concerning the second question, we observe first that there are substantial new issues that need to be considered prior to an implementation and part of the objective of the ensuing sections is to study these issues and to suggest mechanisms for dealing with them. These discussions are thus relevant even if a metaprogramming approach is to be used. We further note that certain situations arise in the processing of our language that are alien to the setting of Horn clauses. These include the introduction of new constants through universal quantifiers and the possibility of sharing variables between clauses and goals. A metaprogramming approach does not offer any natural advantages in dealing with these situations and, therefore, we feel that the specific approach that is adopted here is justified.

4 AN ABSTRACT INTERPRETER

We deal first with the problem arising from existential and universal quantifiers appearing in mixed order in goals. We describe in this section an abstract interpreter for our language that incorporates a solution to this problem within it. The source of the problem is that the set of terms available at the point where a logic variable is introduced may be different from that in existence at a later stage in the computation and that the substitutions that are made for the variable must be restricted to the former set. For example, let us suppose that our language has one unary function symbol f and one constant symbol a and then consider the attempt to solve the goal $\exists x\, \forall y\, p(x, y)$. Using the steps outlined in the previous section, this results in an attempt to solve the goal $p(X, c)$ where c is a new constant. Notice that at this stage our universe of terms has been expanded by the addition of the constant symbol c. However, it is the old collection of terms, that obtained by using f and a and variables, that determines acceptable substitutions for X.

A naive approach to ensuring that only legitimate instantiations are considered for logic variables involves tagging each of these variables with the set of constants that are permitted to appear in terms instantiating it. This set can then be used in an "occurs-check" during unification in order to determine the acceptability of proposed substitutions. Fortunately, the different sets of constant symbols constitute a hierarchy of universes and a practical realization of this idea can be obtained by using a numerical tag with each constant and logic variable. The level 1 universe consists of all the constant symbols that appear in the program clauses and the original goal. These symbols may be tagged by 1 to indicate their position in the hierarchy. Each time a universal quantifier is encountered, a new constant must be introduced, giving rise to the next universe in the hierarchy. This requirement can be accounted for by increasing the "universe index" by 1 and introducing a new constant tagged with this index. The collection of constants at the new level thus consists of all those constants tagged with a number less than or equal to that level. When an existential quantifier is encountered, it is instantiated by a logic variable. This variable may be tagged with the current value of the universe index, the intended interpretation of the tag being that a term may be substituted for the variable only if all the constants appearing in the term have a smaller or identical tag.

The actual use of the tags occurs in the course of unification and consists of the following. The process of unification culminates with an attempt to instantiate a logic variable with a term. In the present context, this would amount to an attempt to set a variable X with a tag i to a term t. Before such an instantiation is permitted, a consistency check must be performed on tags in addition to the usual occurs-check: it must be determined that t does not contain any constants with a tag value greater than i. Actually, one additional device must be incorporated into this basic scheme to make it work correctly. Suppose that we have determined that it is acceptable to set X to t. Before actually doing this, it is necessary to change to i the tags on variables appearing in t that have a value greater than i. This is required in order to prevent a later instantiation of these variables from violating the restrictions on instantiations for X. As an illustration, suppose that our program consists of the single clause $\forall z\, (q(z) \supset p(d(z)))$ and that we are trying to solve the goal $\exists x\, \forall y\, (q(y) \supset p(x))$. After the quantifiers are processed, the goal becomes $(q(c^2) \supset p(X^1)$; we assume that numerical tags are associated with constants and logic variables in the manner just described and we depict these tags as superscripts on the relevant symbols. The attempt to solve this goal results, in turn, in an attempt to solve $p(X^1)$ from a program containing the clauses $q(c^2)$ and $\forall z\, (q(z) \supset p(d(z)))$. Backchaining on the second clause now yields the goal $q(Z^1)$, which fails. The important point to note here is that failure is dependent on the tag value of X being communicated to the new logic variable Z that is used to instantiate the quantifier in the clause in question.

The above discussion outlines a notion of labelled or tagged unification that is relevant to the implementation of the language that is based on *fohh* formulas. A formal presentation of this notion and a study of some of its properties may be found in [18]. For our present purposes it suffices to note that this form of unification can be explained in a fashion similar to first-order unification, that the notion of most general unifiers makes sense in this context as well and that such unifiers can be found by a process identical to that in the usual first-order case except for the checking of tag constraints and the propagation of tags described above. In the sequel, we relativize all the terminology pertaining to unification to this notion in the extended sense just described.

The use of the ideas described above in dealing with a mixture of quantifiers in goals calls for a method for associating tags with the constants and logic variables appearing in such goals. We have already described the way in which this tag is determined if the constant or logic variable is introduced as a result of processing a quantifier. However, processing may start with a goal that already has constants and free variables (that eventually become logic variables) in it. In this case a *tagged version* of the goal is produced by associating the tag 1 with these constants and variables. Similarly, it may be necessary at some point in the computation to create an instance of a program clause and the constants and free variables appearing in such an instance must be tagged. An instance of this kind will be needed when the universe index is at some value I and it constitutes a *new tagged instance* of the clause relative to I that is obtained

(i) by associating the tag 1 with each untagged constant appearing in the clause if the clause is of the form A or $G \supset A$, and

(ii) by picking a new variable w, associating the tag I with w and obtaining a new tagged instance of $[w/x]D$ relative to I if the clause is of the form $\forall x\, D$.

We assume here that the free (alternatively, logic) variables that appear in a program clause are already tagged. This property holds trivially for all the clauses in the original program since these are assumed to be closed and can be seen to hold for all the clauses that arise in the course of the processing that is described below.

We now present the promised abstract interpreter. We note that, in this presentation, the free variables and constants in "goals", the free variables in "program clauses", and the variables and constants in "substitutions" will all be tagged. We continue to refer to these objects as goals, program clauses and substitutions, despite this change. Now, the possibility for implications to be present in goals makes it necessary to consider explicitly the program clauses that are available when a particular goal is being solved. Similarly, the inclusion of universal quantifiers requires the solution of a goal to be parameterized by a universe index. Thus, our abstract interpreter will deal with tuples of the form $\langle G, \mathcal{P}, I\rangle$, where G is a goal, $\mathcal{P}$ is a program and I is a natural number. We shall refer to a multiset of such tuples as a decorated goal set. Let $\mathcal{G}$ be a decorated goal set and let θ be a substitution. Then the abstract interpreter may transform the pair $\langle \mathcal{G}, \theta\rangle$ according to the following rules:

(1) If $\mathcal{G}$ is $\mathcal{G}' \cup \{\langle G_1 \wedge G_2, \mathcal{P}, I\rangle\}$, then by obtaining $\langle \mathcal{G}' \cup \{\langle G_1, \mathcal{P}, I\rangle, \langle G_2, \mathcal{P}, I\rangle\}, \emptyset\rangle$.

(2) If $\mathcal{G}$ is $\mathcal{G}' \cup \{\langle G_1 \vee G_2, \mathcal{P}, I\rangle\}$, then by obtaining $\langle \mathcal{G}' \cup \{\langle G_i, \mathcal{P}, I\rangle\}, \emptyset\rangle$ for $i = 1$ or $i = 2$.

(3) If $\mathcal{G}$ is $\mathcal{G}' \cup \{\langle \exists x\, G, \mathcal{P}, I\rangle\}$, then by obtaining $\langle \mathcal{G}' \cup \{\langle [w/x]G, \mathcal{P}, I\rangle\}, \emptyset\rangle$, where w is a new variable whose associated tag is I.

(4) If $\mathcal{G}$ is $\mathcal{G}' \cup \{\langle (D_1 \wedge \ldots \wedge D_n) \supset G, \mathcal{P}, I\rangle\}$, then by obtaining $\langle \mathcal{G}' \cup \{\langle G, \mathcal{P} \cup \{D_1, \ldots, D_n\}, I\rangle\}, \emptyset\rangle$.

(5) If $\mathcal{G}$ is $\mathcal{G}' \cup \{\langle \forall x\, G, \mathcal{P}, I\rangle\}$, then by obtaining $\langle \mathcal{G}' \cup \{\langle [c/x]G, \mathcal{P}, I+1\rangle\}, \emptyset\rangle$, where c is a new constant whose associated tag is $I+1$.

(6) If $\mathcal{G}$ is $\mathcal{G}' \cup \{\langle A, \mathcal{P}, I\rangle\}$ and $G \supset A'$ is a new tagged instance relative to I of a clause in $\mathcal{P}$ such that A and A' have the most general unifier σ, then by obtaining $\langle \sigma(\mathcal{G}' \cup \{\langle G, \mathcal{P}, I\rangle\}), \sigma\rangle$.[5]

(7) If $\mathcal{G}$ is $\mathcal{G}' \cup \{\langle A, \mathcal{P}, I\rangle\}$ and σ is a most general unifier of A and a new tagged instance of a clause in $\mathcal{P}$ relative to I, then by obtaining $\langle \sigma(\mathcal{G}'), \sigma\rangle$.

The symbol $\cup$ used in these transition rules denotes multiset union. The abstract interpreter for our language now functions as follows. In attempting to solve a goal G given a program $\mathcal{P}$, it will start off with the tuple $\langle \{\langle G', \mathcal{P}, 1\rangle\}, \emptyset\rangle$, where G' is a tagged version of G, and will transform this tuple by repeated applications of the rules above. It will succeed if it eventually manages to obtain a tuple of the form $\langle \emptyset, \theta\rangle$. In this case, the sequence of tuples $\langle \mathcal{G}_i, \theta_i\rangle_{1 \le i \le n}$ that constitutes a successful run for the interpreter is referred to as a *derivation* of G from $\mathcal{P}$, and $\theta_n \circ \ldots \circ \theta_1$ is referred to as the associated answer substitution.

There is an evident non-determinism in the interpreter. This non-determinism can be factored into two forms. First, there may be a choice concerning the tuple from the decorated goal set that is to be processed next. Second, there may be a choice concerning the disjunct that is to be solved if the tuple picked pertains to a disjunctive goal and the program clause that is to be used if the tuple picked pertains to an atomic goal. The latter kind of non-determinism is one that we have discussed already and is manifest in the transition rules (2), and (6) and (7) respectively. The former kind of non-determinism is inconsequential. The following proposition attests to this fact and also verifies the correctness and the adequacy of the abstract interpreter that is described above. A proof of this proposition may be found in [18].

Proposition 4.1 *Let $\mathcal{P}$ be a program and let G be a goal.*

(1) If there is a derivation of G from $\mathcal{P}$ with answer substitution θ, then there is a proof in intuitionistic logic for $\theta(G)$ from $\mathcal{P}$.

(2) If, for some substitution σ, there is a proof in intuitionistic logic of $\sigma(G)$ from $\mathcal{P}$, then there is a derivation of G from $\mathcal{P}$ with an answer substitution θ that is more general than σ. Furthermore, such a derivation can be obtained by picking the next tuple to be processed in an arbitrary fashion.

The abstract interpreter has several features that makes it amenable to a WAM-like implementation. The essential non-determinism that is present in it is similar to that in the case of Horn

[5] Applying a substitution to a set is to be interpreted as applying it to each element and applying it to a tuple corresponds to applying it to each formula that appears in it. Note also that the presence of quantifiers in formulas may require renamings to carried out in the course of applying a substitution. In an actual implementation the representation of free variables eliminates the usual capture problems and thus obviates renaming.

clause logic and can be handled, as usual, by a depth-first search with backtracking, to be implemented through the use of choice point records. In contrast to the Horn clause case, constants and variables have to be tagged and these tags have to be utilized in determining unifiers. The universe index will be needed in the generation of these tags and we add a register called the `UI` register to those present in the WAM for maintaining this index. This register will be manipulated by universal goals, being incremented on entry and decremented on successful completion. Backtracking may, in general, cause a switch to a context embedded within a different number of universal quantifiers and the register must be reset to the appropriate value in such cases. To facilitate such a resetting, choice point records include, in our context, an additional field called `UIP` into which the value of the `UI` register is stored at the time of creation of the record. The use of tags in unification is, of course, quite straightforward. From the perspective of compilation, instructions for unification need to be modified so as to utilize the tags in the required fashion. Although no new instructions are needed for compiling unification, some new instructions are required for handling the effects of universal quantifiers. The details of these aspects are discussed in Section 7.

There is, however, one aspect of implementation that needs further consideration. The possible occurrence of implications in goals requires that the solution of each goal be relativized to a program context. In the abstract interpreter, this requirement is fulfilled by decorating each goal with its program context. Construing such a decoration naively will obviously not lead to an acceptable implementation. However, it is possible to provide a stack-based realization of changing program contexts and we discuss this issue in the next section.

5 DEALING WITH IMPLICATION GOALS

An invocation of the implication goal $D \supset G$ causes D to be added to the program before an attempt is made to solve G and to be removed from the program upon a successful completion of this attempt. Thus, implication goals conceptually entail "asserting" and "retracting" program clauses. Invocations of such goals can be nested inside one another and several layers of these operations may, therefore, have to be performed during execution. However, the assertion and retraction of program clauses follows a stacking discipline and can, in principle, be implemented using a run-time stack.

An actual implementation of the above conceptual model must include devices for dealing with certain additional aspects. One of these aspects is the sharing of code across different versions of the "same" program clause that may be added to the program in the course of solving a query. To understand what exactly is at issue, let us suppose that our program contains the clauses $p(a)$ and $\forall x\,(((D \supset G) \wedge p(x)) \supset p(f(x)))$, where p is a predicate name, f is a function symbol, D is a program clause and G is a goal, and then consider solving the goal $\exists y\, p(f(f(y)))$. The goal $(D \supset G)$ will be invoked twice in the course of solving the given goal. The clause D will, therefore, have to be added to the program twice. However, a satisfactory implementation should maintain only one copy of the "code" for D and use this in realizing both additions. Adopting this approach is necessary both for controlling the sizes of program and for supporting the compilation of program clauses.

The sharing of code for program clauses can be easily accomplished in the example considered above: we simply maintain one copy of the code for D and use pointers to this on the two different occasions that it is added to the program. However, as discussed in Section 3, it is possible for program clauses to contain free variables and so this idea does not quite solve the problem in the general case. As a specific illustration, suppose that the second program clause in the example considered above is replaced by the clause

$$\forall x\,(((D(x) \supset G) \wedge p(x)) \supset p(f(x)));$$

we assume here that $D(x)$ represents a program clause with x occurring free in it. The two program clauses that are added to the program in the course of solving the goal $\exists y\, p(f(f(y)))$ are now $D(f(y))$ and $D(y)$. These program clauses are, in a sense, distinct. However, they could have a considerable amount of structure in common, and a reasonable implementation scheme should permit this structure to be shared.

The above considerations lead naturally to a representation of a program clause as a composite of (a pointer to) code and a set of bindings for its free variables. Such a representation corresponds to the idea of a closure that is used in implementations of functional programming languages and is an enrichment to the usual WAM treatment of program clauses. Using such a representation makes it possible to compile both the program clauses that appear as the antecedents of implications and the action to be taken on encountering implication goals. In understanding how this might be done, let us assume that programs are maintained as lists of closures that are searched sequentially in order to determine the clauses relevant to solving given atomic goals; this representation of programs differs from the one used in the WAM and is also extremely naive, but we defer the consideration of more sophisticated representations till the next section. Now suppose that there is an occurrence in the original program or goal of an implication goal of the form

$$(D_1(\bar{x}_1) \wedge \ldots \wedge D_n(\bar{x}_n)) \supset G,$$

where, for $1 \leq i \leq n$, $D_i(\bar{x}_i)$ denotes a program clause and $\bar{x}_i$ is a listing of the variables occurring free in it. The variables in $\bar{x}_i$ are, in fact, ones that are bound by quantifiers that surround the implication goal in question. An invocation of this implication goal will, therefore, take place in a context where these variables have been replaced by logic variables or by generated constants. As we shall see in detail in Section 7, bindings for these variables at a particular invocation can be given by compile-time determined offsets relative to the current environment record. Now, the program clause given by $D_i(\bar{x}_i)$ can be compiled in the usual fashion with the exception that it should include instructions for initializing the variables in $\bar{x}_i$ as described below. Let c_i be a pointer to this code. The enhancement to the program that is required at an invocation of the implication goal can be realized by adding to it the closures $\langle c_i, e\rangle$ for $1 \leq i \leq n$, where e is a pointer to the current environment record. This action can itself be compiled by statically associating with the goal a table of pointers to code for the program clauses that constitute its antecedent. Suppose now that a version of the clause $D_i(\bar{x}_i)$ that is given by the closure $\langle c_i, e\rangle$ is invoked in an attempt to solve some (sub)goal. The code that is to be executed is pointed to by c_i. This code must, first of all, relativize the bindings for the variables in $\bar{x}_i$ to the environment record created for the invocation. Doing this involves copying over the bindings for these variables in the environment record pointed to by e. As we have already noted, the offset for each of these variables relative to the old environment record can be statically determined and so the necessary initialization process can itself be compiled.

A second aspect to which special attention must be paid in the presence of implication goals is the context switching necessitated by backtracking. The broad requirement upon backtracking is to reinstate a program that was in existence at some earlier point in the computation. To understand

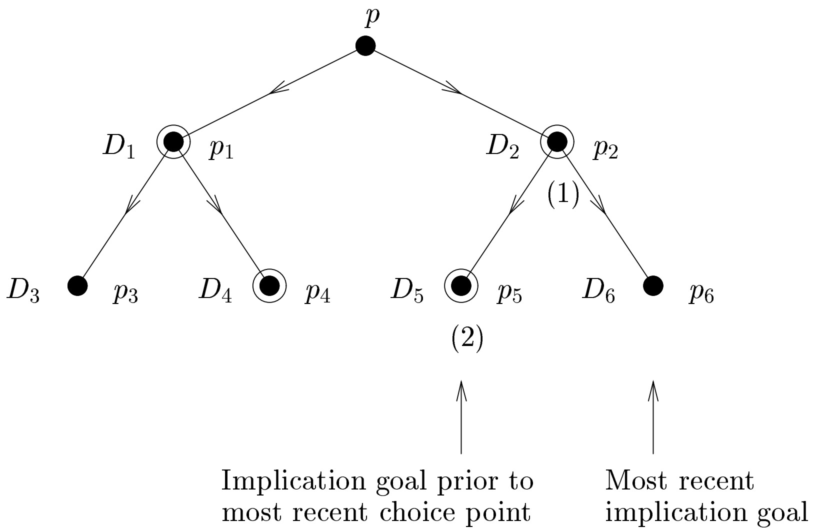

Figure 5.1: Example of an Implication Tree

clearly the changes that must be effected, and consequently the bookkeeping that must be done, let us consider a program containing the clauses

$$((D_1 \supset p_1) \wedge (D_2 \supset p_2)) \supset p,$$
$$((D_3 \supset p_3) \wedge (D_4 \supset p_4)) \supset p_1,$$
$$((D_5 \supset p_5) \wedge (D_6 \supset p_6)) \supset p_2$$

and possibly others defining the predicates p_3, p_4, p_5 and p_6; we assume that $D_1, \ldots, D_6$ represent program clauses and that $p, p_1, \ldots, p_6$ are predicate names. Suppose now that an attempt is made to solve the goal p. This attempt engenders the invocation of implication goals whose dynamic nature can be represented by a tree-like structure that we call an *implication tree*. For instance, let us assume that the first clause above is being used in the attempt to solve p, that, in this context, a solution to $D_1 \supset p_1$ has been found by using the second clause to solve p_1 and that an attempt is now being made to solve p_2 after having augmented the program with D_2. Let us further assume that the third clause is being used in the attempt to solve p_2, that $D_5 \supset p_5$ has already been solved in this attempt and that a solution for p_6 is being sought from a program that additionally contains the clause D_6. The state of the computation as it relates to the invocation of implication goals can then be depicted by the tree shown in Figure 5.1. The nodes of this tree, with the exception of the root, correspond to the invocations of implication goals, the clause that is added as a result of this invocation being shown to the left of the node and the goal that is subsequently invoked being shown to the right. The root of the implication tree represents the original goal. The left-to-right ordering of the nodes reflects the time order of subcomputations and the circles that are drawn around some nodes indicate the existence of choice points subsequent to the goal invocation that they represent.

Suppose now that, in the situation being considered, the attempt to solve the goal p_6 fails. The next step in the computation must then be an attempt to find an alternative solution for the most recent prior goal for which such a possibility exists. Referring to the implication tree shown in Figure 5.1, this means that a goal that lies below the node labelled (2) must be returned to. The attempt to find another solution to this goal must, of course, be preceded by a reinstatement of the program that was in existence when the first (successful) attempt to solve it was made. This earlier program state can be easily recreated if the implication tree is available: We find, first of all, the closest common ancestor in the implication tree of the node representing the most recent implication goal, *i.e.*, the latest implication goal below which the failure has occurred, and the node denoting the last implication goal below which the most recent choice point exists. Referring again to Figure 5.1, this closest common ancestor is the node labelled (1). The desired program is then obtained by discarding all the program clauses that were added along the path from this node up to and including the node representing the most recent implication goal, and adding back all the program clauses that had been added (and subsequently discarded) along the path from this node up to and including the node denoting the last implication goal prior to the most recent choice point. In the example being considered, this translates into discarding the program clause D_6 from the program and adding back D_5.

Implementing the context switching process described above, requires a record of the (annotated) implication tree and the nodes in this tree representing the most recent implication goal in a global sense and relative to each choice point to be maintained at each stage of computation. We propose maintaining the implication tree by creating an *implication point* record on the local stack at the start of the computation and each time an implication goal is invoked. In order to describe the structure of this record, we need to be more concrete about the representation of the programs that are in existence at various points in computation. Recall that we intend to maintain these as lists of closures. We shall assume for the moment that new closures are added at the end of this list. The starting point of this list is, therefore, fixed and the program available at any stage is specified completely by the end point. We add a register named `LC` to the usual WAM registers for recording this end point relative to any point in computation. Taking this representation of programs into account, an implication point record will have the following information fields:

(i) a reference, `IC`, to the statically constructed table of pointers to code for the clauses that constitute the antecedent of the implication goal,

(ii) a pointer, `E'`, to the environment record that is current at the invocation of the implication goal that the implication point record corresponds to,

(iii) a pointer, `IP`, to the implication point record for the most recent implication goal within which the implication goal corresponding to the implication point record in question appears embedded, and

(iv) a pointer, `LCP`, to the end of the list of closures constituting the program at the time the implication goal is invoked.

Note that the start of the computation can itself be viewed as the invocation of an implication goal whose antecedent is the original program and whose consequent is the corresponding goal and the first implication point record will be set up consistent with this viewpoint and with its `IP` field indicating that there is no enclosing implication goal invocation. Now, the nodes in the implication tree are obviously represented by implication point records. Thus, the remaining information that must be maintained for context switching purposes consists of the most recent implication point record relative to the current point in computation and to each choice point. The former is maintained in another new register that is named `I`. For the latter, we add a field called `IP` to the choice point record of the WAM; this field will be set to the value of the `I` register at the time the choice point record is created.

We can now describe a scheme that accounts completely for implication goals. Under this scheme, the invocation of an implication goal causes an implication point record to be created and additions to be made to the existing program. The augmentation to the program is carried out in an obvious way using the `E` register of the WAM and the table generated at compile-time for the implication goal. The `IC` field of the implication point record is set simply to point to this table and the contents of the `E`, `I` and `LC` registers prior to the invocation of the implication goal determine the `E'`, `IP` and `LCP` fields. At the end of this process, the `I` register is updated to point to the newly created implication point record. Note also that the `LC` register will be affected by the augmentation to the program. When an implication goal is successfully completed, the `I` and `LC` registers must be reset to their values prior to the invocation of the goal. This is done by using the relevant fields from the implication point record corresponding to the goal that will be given by the contents of the `I` register. The implication point record can also be discarded if the implication goal is more recent than the most recent choice point; this can be determined simply by comparing the `I` and the `B` register (that indicates the location in the stack of the most recent choice point record as in the WAM) and, thus, the top of the local stack will be given by the largest of the addresses in the `I`, `E` and `B` registers. Finally, suppose there is a failure at some point in the computation. Assuming that there is an alternative solution path to be explored, the appropriate program context is recreated as follows: We first check if the `I` register points to a location earlier on the stack than the one pointed to by the `B` register. If so, the program context is already the appropriate one. Otherwise, we chain back through implication point records starting from the record pointed to by the `I` register till we reach one that appears lower in the stack than the location pointed to by the `B` register. Let us refer to this implication point record as CCA (for closest common ancestor). We then discard the necessary closures by setting the `LC` register to the value of the `LCP` field stored in the implication point record just prior to CCA on the path to it from the record pointed to by the `I` register. The `I` register is then set from the `IP` field of the most recent choice point record. Finally, the addition of the relevant closures is affected by using the `IC` and `E'` fields of the implication point records between CCA and that given by the `I` register. The `LC` register must, of course, be updated at the end of this process.

We have assumed above that the addition of program clauses that results from invoking an implication goal takes place at the end of the program. Where exactly the addition should occur is unspecified by the semantics that we have presented for the language. It is conceivable, perhaps even desirable, that this addition be recorded at the beginning of the program, thereby making the new clauses accessible before the ones already in the program. The model described above is amenable to this interpretation; it is the beginning of the list of closures that must be recorded in this case instead of the end.

6 AN EFFICIENT REALIZATION OF PROGRAM CONTEXTS

The list representation of programs adopted in the last section is useful from the perspective of understanding the addition and deletion of program clauses. However, it is not adequate from a practical standpoint since it does not allow rapid access to program clauses: the list must be searched sequentially to determine the applicable clauses. We can minimize the cost of this search by maintaining a hash-table that is accessed by the name of the predicate. Each entry in the hash-table points to a list of lists, one list for each distinct predicate name that hashes to the same table entry. The list associated with each predicate contains a closure for each program clause that may be used in solving an atomic goal whose name matches the predicate. Additions to a list may occur either at the end of this list or at the beginning, depending on the chosen semantics. In lieu of the pointer into the original list of program clauses kept in each implication point record, we must now maintain pointers into the closure lists for *each* predicate that is defined in the antecedent of the implication goal. These pointers facilitate the discarding and reintroduction of closure entries upon successful completion of an implication goal and upon backtracking. If the antecedent of an implication goal contains multiple clauses for a predicate, this scheme can be modified to accommodate the compilation of sequencing with respect to these clauses.

In the case when new clauses are added to the front of a program, this general idea can be implemented in a fashion that enables the context switching required on backtracking to be realized with a minimum of effort. The starting point for this scheme is an organization of the global program, *i.e.*, the program in existence prior to the user's query, in a form that supports rapid access to the code for any given predicate. In particular, we assume that multiple clauses for a predicate give rise to one procedure with several entry points as in the WAM and that this code may be located, for instance, by hashing on the name of the predicate. Now each time an implication goal is invoked, new clauses may be added to the program for any given predicate. In order to provide efficient support for the process of chaining through the clauses for a given predicate that are introduced by different implication goals, we construct an access vector of pointers that effectively identify the next most recent set of clauses for the predicates defined by the clauses in the antecedent of the implication goal. This access vector is computed at the time the implication goal is invoked and is saved in the implication point record that is created for the invocation. This implication point record will be retained even after a solution to the corresponding goal has been found so long as backtracking may cause the goal to be retried. Thus the access vector will be available too, and the old program context can be resurrected simply by reverting to its use in finding clauses for solving goals.

In spelling out the details of the scheme outlined above, we make use of the discussions of the previous section. The central point, as before, is the treatment of implication goals that appear in the program. Once again, let

$$(D_1(\bar{x}_1) \wedge \ldots \wedge D_n(\bar{x}_n)) \supset G$$

be a schematic representation of such a goal. Now, each of the clauses $D_i(\bar{x}_i)$ will be compiled in the manner described in the previous section. In contrast to the earlier situation, however, clauses that define the same predicate will be combined into *one* procedure with the use of clause sequencing code. In general, this will give rise to m segments of code, defining m predicates. In conjunction with the implication goal, a table will be created at compile-time with the following entries:

(i) The number of predicates defined in the antecedent of the implication goal, *i.e.*, m in the case considered. We call this the *size* of the implication goal.

(ii) A pointer to code that, given any predicate name, either determines that it is not defined by any of the clauses in the antecedent of the implication goal or returns the location of the relevant compiled code. The structure of the code that carries out this task will depend on the number of predicates that are defined in the antecedent: if this is a small number, then sequential search will suffice; otherwise, a hash-table may be used.

(iii) A one-to-one mapping from the names of the predicates defined in the antecedent of the implication goal to $\{1, \ldots, m\}$. We refer to the number associated with a particular name as its *offset number* relative to the implication goal. This mapping is needed in setting up implication point records as we explain presently.

As mentioned above, access to program clauses will be provided through implication point records in the new scheme. This access is realized, at a conceptual level, as follows. The search for clauses defining a particular predicate takes place relative to an implication point record that is pointed to by a new register called the `CI` register. At the outset, *i.e.*, when an atomic goal is encountered, this register is set to point to the most recent implication point record by copying into it the contents of the `I` register. If `CI` points to an implication point record representing the invocation of an implication goal whose antecedent does not contain a clause defining the predicate in question, then `CI` will be updated to point to the implication point record for the closest dynamically enclosing implication goal invocation and the search will continue from there. If there is no such enclosing implication goal, a failure will result. (As before, we view the start of computation as the first invocation of an implication goal, one for which an implication point record is created at the bottom of the stack.) On the other hand, if there are clauses defining the predicate in the antecedent of the relevant implication goal, then these will be used in an attempt to solve the atomic goal. The use of these clauses will, in general, require bindings for certain variables to be initialized from an appropriate environment record. The location of this environment record is available from the implication point record pointed to by `CI` and will be copied into another new register called `CE` before the clauses in question are used.

Now suppose that all the clauses available for a predicate through a particular implication point record have been tried and have resulted in failure. There may still be clauses available that can be tried in an attempt to solve the given atomic goal. Conceptually these clauses can be located by chaining back through implication point records for the enclosing implication goals. However, this work can be reduced by doing it once and for all at the time the implication point record is created. In particular, for each predicate defined in the relevant implication goal, we can compute and store a pointer to the closest enclosing implication point record containing a clause for that predicate and a pointer to the corresponding code. This optimization, while useful in general, turns out to be particularly helpful in compiling clause sequencing as we shall see in the next section.

Taking the various discussions of this section into account, the information that is now to be stored in an implication point record is the following:

(a) A pointer, `IC`, to the code for determining whether a predicate is defined by any clauses in the antecedent of the corresponding implication goal and, if it is, for finding the address of the code generated from these clauses. The address to be stored in `IC` is available from the table compiled for the implication goal; see item (ii) above. (This field is conceptually similar to one of the same name in the implication point record of Section 5.)

(b) A pointer, `E'`, to the environment record that is current at the invocation of the implication goal that the implication point record represents.

(c) A pointer, `IP`, to the implication point record for the most recent implication goal within which the implication goal corresponding to the implication point record in question appears embedded.

(d) An access vector, nc, whose size is that of the implication goal. The ith entry of this vector contains a pointer to the code for the next clause for the predicate with offset number i and a pointer to the implication point record in which this clause "occurs"; if such a clause does not exist, the address of a failing procedure is inserted.

The last component is computed at the time the implication point record is created.[6] The manner in which this computation is carried out should be obvious from the previous comments.

The availability and interpretation of the code for clauses within the scheme outlined is dependent on the values of the `CI` and `CE` register. Consequently these registers must be saved in the choice point record. Accordingly, our choice point records contain three new fields in comparison with the WAM, the `IP`, the `CIP` and the `CEP` fields. We also observe the ease with which the program context can be reset to the required value upon backtracking: the current program and the clauses yet to be tried in solving a particular atomic goal are determined by the value of the `I` and `CI` and `CE` registers respectively, and it is only necessary to set these registers from the corresponding fields in the most recent choice point record.

It is useful to understand qualitatively the cost of the proposed scheme for supporting a scoping ability relative to program clauses. At the very outset, we note that merely having the flexibility of changing the program context dynamically incurs an overhead even if it is not used, *i.e.*, even if the program consists solely of program clauses from the Horn clause setting. There are two sources for this overhead. First, each choice point record must store three extra fields — the `IP`, `CIP` and `CEP` fields — with associated space and time costs. Second, the location for the code to be used in solving a given atomic goal can only be determined dynamically, perhaps via a hash-table. Allowing for universal goals adds one more field, the `UIP` field, to choice point records and incurs a cost for tagged unification whose precise nature will become evident in the next section.

Certain costs are incurred in addition to those above if a genuine use is made of the scoping ability relative to program clauses that is afforded by our language. First of all, the search for the code for a predicate becomes more complex. A reasonable assumption for the time required to locate code for a predicate in a program unit, *i.e.*, the block of code corresponding to the original program or the antecedent of an implication goal, is that it is fixed. Viewing the top-level goal itself as an implication goal, the (time) degradation in locating the code for a predicate then depends on the deviation from 1 of the number of nested implication goal invocations within which the attempt to find such code takes place. In assessing the overall degradation, it is necessary to amortize the number of nested implication goal invocations over all procedure calls and also to consider the proportion of all operations that procedure calls constitute. Taking these aspects into account and noting that well-written programs should result in only a small nesting of implication goals, we believe that the overhead due to this factor will be small. A second factor affecting performance

[6] An alternative approach is possible: the access vector may be computed "on demand", *i.e.*, the location of the next clause may be computed when first needed and stored in the vector to facilitate a direct lookup on subsequent occasions.

is the need to set up and maintain implication point records. Let n be the number of predicates that are defined in the antecedent of the implication goal corresponding to an implication point record. The space required for the record is then $3 + 2 * n$ pointers. The only time expenditure in creating the record that is not fixed is that for setting up the access vector. As already noted, the time needed for locating the code for a predicate in a dynamic context is proportional to the number of nested implication goal invocations by which the context is defined. The time required for computing the access vector would be n times this cost. The number of predicates defined in the antecedent of an implication goal and the nesting level of implication goal invocations will, in the typical situation, be bounded by a small number. The overall space and time costs due to this factor will thus be roughly proportional to the number of implication point records that are set up in the course of solving a query. This number can be assumed to be small, especially in comparison with the number of procedure calls and other operations that will have to be performed.

Before concluding this section, we note the similarity between the scheme that we have outlined here and that used for contextual logic programming in [12]. Indeed, the mechanisms presented in this section are an amalgamation of the ideas discussed in Section 5 (and in [10]) and those in [12]. Our scheme differs in detail from that in [12] in that (a) we eliminate the context stack by using implication point records that are stored on the local stack, (b) we need to deal with closures instead of just program code, and (c) implication goals involve only one of the several semantics that are implemented in [12].[7]

7 COMPILATION

A scheme for compiling a logic programming language into WAM-like instructions must address two main issues: the compilation of unification and the compilation of control. The same general approach can be used in a treatment of these aspects in the context of our language as in the case of Prolog. However, there are differences in detail, arising from the fact that some new problems have to be handled in an implementation of our language. We have presented schemes for dealing with these problems in earlier sections and have also indicated the possibility of compilation within these schemes. We provide concreteness to the latter discussion in this section by describing modifications and additions to the instructions of the WAM for accounting for (a) the tagged form of unification, (b) the larger variety of non-atomic goals, and (c) the possibility that the clauses that appear in the antecedents of implication goals actually extend previously existing definitions of predicates. We also illustrate the use of the resulting instruction set in compiling programs in our language.

7.1 Compilation of Unification

We shall assume that the set of instructions that are included in the WAM for the purpose of compiling unification is that described in [1] as opposed to the one contained in [26]. The main difference between these two sets is that the former includes a collection of `set` instructions that parallel the `unify` instructions. These `set` instructions are used instead of the `unify` instructions

[7] An implication goal $D \supset G$ can be interpreted as the goal $U >> \bar{G}$ in contextual logic programming where U is a unit containing a translated version of the program clauses in D and $\bar{G}$ is the translation of G obtained by using this transformation recursively. Under this interpretation, the operational semantics we have defined for implication goals corresponds in contextual logic programming to assuming that the clauses in a unit extend the definitions of predicates available in a dynamic context and that goals are solved by using a lazy binding in the sense of [12].

in compiling the creation of terms in the scope of the `put_structure` and `put_list` instructions. While this "enhancement" to the instruction set is not essential, it is useful in reducing mode setting and testing in the context of the WAM and also provides the basis for avoiding some occurs-checking and the checking of tag compatibility in our context. Now, despite the changed nature of unification for our language, no instructions are needed in addition to those already in the WAM for implementing this operation. However, some of the WAM instructions must be modified to ensure that tags are maintained and respected during unification.

The tagging of variables is dependent on their classification as either temporary or permanent. This classification must be performed relative to each program clause whose compilation is to be considered, *i.e.*, relative to each clause that is part of the original program or that appears as one of the conjuncts in the antecedent of an implication goal. The variables of such a clause that need to be classified as temporary or permanent are the following: (a) those that are free in the clause — this is relevant only in the case that the clause appears in the antecedent of an implication goal, (b) those that are (implicitly) universally quantified over the clause, and (c) those that are explicitly quantified in the body of the clause but where the quantification is not embedded in the antecedent of an implication goal. Of these variables, those that are universally quantified in the body of the clause or have an occurrence in the antecedent of an implication goal appearing there are considered permanent. An existentially quantified variable or a variable of category (b) is also considered permanent if it has an occurrence in a universal goal that appears within the scope of the quantifier governing the variable. The categorization of the remaining variables is determined after the given clause is reduced to one in the Horn clause setting by dropping quantifiers in its body and replacing implication goals by their consequents. Free variables, *i.e.*, variables of category (a), are considered temporary if their occurrences are limited to the head and first goal in the body under this reduction and permanent otherwise. Finally, for the rest of the variables we use the classification employed with the WAM, with the proviso that a goal that originally appeared embedded inside an implication or a universal quantifier is not to be considered a last goal under the reduction.

No tags are associated with temporary variables initially; tags for these variables are determined by the instructions that manipulate them as we see below. The permanent variables of a clause are tagged with the value of the universe index at the time the clause is invoked. The relevant tag value is obtained from the `UI` register and the tagging action is carried out by the `allocate` instruction. This instruction is, in our case, provided with an argument indicating a number of suitably tagged unbound references that are to be created on the top of the stack. This action makes unnecessary the initialization that is performed by `put_variable` and `set_variable` relative to permanent variables. These instructions can therefore be eliminated and the `put_value` and `set_value` instructions can be used in their place.

The unification related instructions are changed in the following fashion. The instructions that write constants must now also associate the tag 1 with these constants. This requirement affects the instructions `put_constant`, `set_constant` and, in the appropriate contexts, `get_constant` and `unify_constant`. Instructions that bind or create variable cells must, similarly, be sensitive to tag associations. Among these, it turns out that the instructions `get_variable` and `unify_variable` (and `unify_void`) executed in *read mode*, do not need to handle tags at all; the binding will always be permitted and the incoming structure, variable or constant will carry the necessary tags. The `set_variable` and `set_void` instructions must tag the variable cells that they create on the heap with the value of the `UI` register. The `put_variable` instruction, used now only with respect to a temporary variable, must perform a similar association. The instructions `put_unsafe_value` and `set_local_value` create new variable cells on the heap in certain situations and, in these cases, they must associate the tag value of the stack variable that is being "copied" with the newly created cell. Finally, when the `unify_variable` and `unify_void` instructions are executed in *write mode*, they must associate a tag value with the variables being written that is equal to the tag value of the variable whose value is being set by the governing `get_structure` instruction.[8] To facilitate the communication of this tag value between the `get_structure` and the `unify_variable` instructions, use is made of a new register called the `UT` register. The `get_structure` instruction copies the relevant tag value into this register when it encounters an incoming argument that is an unbound variable.

The instructions considered up to this point only require modifications to ensure that the right tag values are written with variables and constants. The only times at which the compatibility of tags need to be checked are when two constants are being matched by `get_constant` or `unify_constant` and within the unification process that is carried out in interpretive mode in conjunction with `get_value`, `unify_value` or `unify_local_value`. In the former case, the check that must be made amounts simply to considering tag values to be parts of the names of constants. In the latter case, the necessary check causes variable assignments to be constrained according to the following: A variable cannot be bound to a constant with a higher tag or to a structured term containing a constant with a higher tag. If a variable is bound to another variable with a higher tag or to a structured term containing a variable with a higher tag, then the tag value of the latter variable must be set to that of the former. A `unify_value` or `unify_local_value` instruction executed in *write mode* is already a part of a variable assignment. The tag value of the variable being assigned to is contained in the `UT` register and it is this value that must be used in the described check of tag compatibility. Note that the interpretive unification process that is being considered must include an occurs-check in an implementation that is sound with respect to the logic considered in this paper or, for that matter, with respect to Horn clause logic. The additional checking of tag compatibility is similarly needed for soundness in the case of our language and can, in fact, be carried out in the same phase as the occurs-check. There is a possibility of avoiding both the occurs-check and the checking of tag compatibility in certain situations and doing so may well be important to the efficiency of an actual implementation. However, a detailed examination of this issue is beyond the scope of this paper.

7.2 Compiling Complex Goals

The issue of concern here is the compilation of the logical symbols that may appear in goals. The symbols in question are $\vee$, $\wedge$, $\forall$, $\exists$ and $\supset$. Goals of the form $G_1 \vee G_2$ and $G_1 \wedge G_2$ are also permitted in Prolog and the method of treatment used there is adequate in our context as well. In particular, $\vee$ gives rise to code for generating a choice point record and $\wedge$ results in the sequential execution of the code for the subgoals.

The treatment of the universal quantifier follows the lines indicated in Section 4. Thus, consider the goal $\forall x\, G$. Bearing in mind the classification of variables described in Subsection 7.1, the variable x would be deemed a permanent variable in the context in which this goal is encountered and so a cell will be allocated for it in the current environment record. Now, the code that is

[8] The comments of Pascal Brisset made us aware of an error with regard to this point in an earlier version of our implementation scheme. The same observation was also made by one of the authors of this paper, Keehang Kwon.

generated for the given goal must increment the `UI` register, place a new constant whose tag value is that contained in the `UI` register in the cell allocated for x and then execute the code for G. Further, if the code for G completes successfully, the `UI` register must be decremented. Three new instructions are introduced for supporting these requirements:

```
incr_universe
decr_universe
set_univ_tag Yi
```

The first two instructions respectively increment and decrement the `UI` register, and the last instruction binds the (permanent) variable `Yi` to a new constant that is tagged with the value of the `UI` register.

A final comment concerning the treatment of universal goals is that, as noted in Section 4, the value of the `UI` register must be stored in the `UIP` field of a choice point record at the time that this record is created.

The action to be performed in conjunction with an existentially quantified goal depends on whether the quantified variable is classified as permanent or temporary. Suppose that the goal that is encountered is $\exists x\, G$. If x is considered to be a permanent variable, then the tag value of the cell allocated for x must be set using the `UI` register and the compiled code for G must be executed. On the other hand, no tags need to be set if x is considered to be a temporary variable and execution can proceed directly to the code for G. In realizing these actions, there is need for only one new instruction:

```
set_exist_tag Yi.
```

This instruction tags the permanent variable `Yi` with the value of the `UI` register.

The treatment of implication goals was discussed in detail in the previous section. Recalling this, when a goal of the form $D \supset G$ is encountered, an implication point record representing the addition of D to the program must be pushed onto the local stack and access to the resulting program must be relativized to this record. In the case that the implication goal completes successfully, access to the program must be restored to being through the implication point record pointed to by the `I` register prior to the invocation of the goal. Compilation of these actions is supported by the following new instructions:

```
push_impl_point t,n
pop_impl_point
```

In the first instruction, `t` represents a pointer to the statically created table for an implication goal that was described in Section 6 and `n` represents the number of variables in the current environment record. This instruction results in an implication point record being pushed onto the top of the local stack, this being located by examining the `I` and `B` registers and the `E` register plus the size of the current environment record. The manner in which the instruction fills in the fields of the implication point record should be obvious from the discussions in the last section. After creating the implication point record, the instruction updates the `I` register to point to it.

The instruction `pop_impl_point` simply restores the previous value of the `I` register by using the `IP` field of the implication point record that the register currently points to.

7.3 Compiling Atomic Goals and Clause Sequencing

The compilation of clause sequencing for the global program, *i.e.*, the program in existence prior to the user's query, remains unaltered from that used for Prolog relative to the WAM. However, there is a slightly different interpretation to the instructions that are used. Those instructions that create a choice point record — specifically, `try_me_else` and `try` — must now also store the contents of the `UI`, `I`, `CI` and `CE` registers in the record. Correspondingly, the backtracking action performed by the instructions `retry_me_else`, `retry`, `trust_me` and `trust` must include a restoration of the values of the `UI`, `I` and `CE` registers from the relevant choice point record.

The clauses in the antecedent of an implication goal are compiled assuming that they constitute a unit, distinct from the rest of the program. The code that is generated for predicates defined in this unit differs from that that would be generated in the case of the global program in only two respects. The first difference is that the code produced for those clauses that have free variables in them will have a part that relativizes the bindings for these variables to the current environment record. The new instruction

```
initialize Vn,m
```

in which `Vn` is a temporary or a permanent variable and `m` is a number is used to achieve this effect. This instruction is like the `get_variable` instruction of the WAM except that the second argument is obtained by using the `m`th variable from the environment record pointed to by the `CE` register. The second difference is that the code that is generated will always contain the creation of a choice point record and its last instruction will be one that has the effect of attempting other clauses that may be available in the dynamic context for the relevant predicate. The instruction

```
trust_ext Pi
```

is added for this purpose.[9] In this instruction, `Pi` is an offset number relative to an implication goal. When the clauses for a particular predicate that appear in the antecedent of an implication goal are compiled, the code for the last clause is preceded by a `try_me_else Li` or a `retry_me_else Li` instruction and is followed by

```
Li :  trust_ext Pi
```

where `Pi` is the offset number for the predicate. Executing this instruction has the following effect: The current choice point record is used to reset all the registers except `P`, which is the program pointer as in the WAM. The entry at location `Pi` in the *nc* field of the implication point record pointed to by `CI` is then used to set the `CI` and `P` registers. Finally, the `CE` register is set to the `E'` field of the record pointed to by `CI`.

With regard to the compilation of an atomic goal, the code for preparing the argument registers follows the pattern used relative to the WAM. The actual invocation of the code for the corresponding predicate name is also achieved through the `call` or `execute` instruction. However, these instructions have a different interpretation in our case from that in the WAM. For example, consider `call q,n`. The search that is made for code for `q` in executing this instruction must depend on the dynamic context. This search starts by setting `CI` to the value in `I` and proceeds in the fashion outlined in the previous section. If code is found, it is executed as described. Otherwise backtracking occurs.

[9] We borrow this instruction from [12].

7.4 Examples of Compiled Code

We adopt below the Prolog conventions of writing implications in program clauses backwards and depicting it by the symbol `:-`, of representing conjunctions in clause bodies by commas and of leaving the top-level universal quantifiers implicit. We present two examples, one illustrating the compilation of multiple clauses in the antecedent of an implication goal that define the same predicate, and the other illustrating the processing of a mixture of quantifiers in goals.

For the first example, we use one of the definitions of `rev` from Section 2:

```
rev(L1,L2) :-
    ((rev_aux([],L2) ∧
      (∀X ∀L1 ∀L3 (rev_aux([X|L1],L3) :- rev_aux(L1,[X|L3]))))
               ⊃ rev_aux(L1,[])).
```

This clause has one permanent variable, namely `L2`. This variable occurs in the first clause for `rev_aux` and the code for that clause must include an instruction for initializing it. We assume that the statically determined table for the implication goal that appears in the definition of `rev` is pointed to by `t1`. The compiled code corresponding to `rev` is then the following:

```
rev:     allocate 1
         get_variable Y1,A2
         push_impl_point t1,1  % add rev_aux code
         put_constant [],A2
         call rev_aux,1
         pop_impl_point        % restore earlier program
         deallocate
         proceed
```

Note that the `call` instruction is used here instead of the `execute` instruction for invoking the `rev_aux` procedure. This invocation appears to be the last call in the body of the clause and it may therefore seem that the code that is shown does not include the last call optimization that is common to Prolog implementations. However, a little thought reveals this not to be the case. The last action that must be performed actually relates to the implication goal that forms the body of the clause: the clauses that are added in the course of solving it must be removed *after* solving `rev_aux`. It might be possible to include this action within the code produced for `rev_aux`, thereby permitting the environment record for `rev` to be discarded before this code is invoked. However, it seems unlikely that doing this will improve space usage significantly. The environment record that is retained at present only contains bindings for tied variables and continuation information and any modified scheme will also have to maintain such information. Furthermore, the main utility of last call optimization is in the context of recursive calls and it is reasonable to assume that such calls will not appear repeatedly in situations where the program is being extended, *i.e.*, embedded within implication goals. We note in this connection that our scheme *does not* affect the usual applicability of last call optimization.

The following code would be generated for the two clauses defining `rev_aux` in the body of the implication goal:

```
rev_aux:  try_me_else C1
          initialize X3,1        % X3 = L2
          get_constant [],A1
          get_value X3,A2        % unify L2 and second argument
          proceed

C1:       retry_me_else C2
          get_list A1
          unify_variable X3
          unify_variable A1
          get_variable X4,A2
          put_list A2
          set_value X3
          set_local_value X4
          execute rev_aux

C2:       trust_ext 1
```

A point to note with respect to this code is that the choice point record is not discarded before the second clause for `rev_aux` is used. The reason for this is that this clause may not be the last one for the predicate in the relevant dynamic context. As observed in Section 2, a universal quantification over `rev_aux` will ensure that this is the case and will, in fact, provide this information to a compiler as well. Such a quantification is not permitted in the language currently being considered, but is included in the extension that we examine in the next section.

The second example that we consider is that of compiling the clause

```
p(Y) :- (∀U ∃Z
              ((∀W (d1(Y,W,Z) :- r(Y,W))) ∧
               (∀W (d2(Z,W) :- d1(Z,W,W)))
                  ⊃ ∃V g(Z,U,Y,V)),
          h(Y).
```

In generating code for this clause, it is necessary to determine the free variables of the clauses that form the antecedent of the implication goal that appears in its body. These variables are those that appear in the relevant clauses and whose (implicit or explicit) quantification governs the implication goal. Thus, the free variables of the clause defining the predicate `d1` are `Z` (explicitly quantified) and `Y` (implicitly quantified) and the only free variable of the clause defining `d2` is `Z`. Bindings for these variables must be contained in the environment record corresponding to `p` at a point when the respective clauses are invoked and it is for this reason that they are deemed permanent variables of the clause defining `p`. The variables `U` and `V` are also permanent variables of this clause and, assuming that `t2` points to the table constructed for the implication goal, the code that would be generated for the clause is the following:

```
p:   allocate 4
     get_variable Y1,A1
```

```
incr_universe         % (∀
set_univ_tag Y2       %   U
set_exist_tag Y3      %    ∃Z
push_impl_point t2,4  %      add clauses for d1 and d2
set_exist_tag Y4      %       ∃V
put_value Y3,A1       %         g(Z,
put_value Y2,A2       %           U,
put_value Y1,A3       %           Y,
put_value Y4,A4       %           V
call g,4              %          ),
pop_impl_point        %      discard clauses for d1 and d2
decr_universe         % ),
put_value Y1,A1       % h(Y
deallocate            %
execute h             %  )
```

We do not present the code for the clauses `d1` and `d2`. The structure of this code should be clear from the previous example.

8 DEALING WITH HIGHER-ORDER ASPECTS

The propositional and quantifier structure of goals and program clauses in the theory of higher-order hereditary Harrop formulas bears a close similarity to that for these formulas in the first-order language considered so far. One distinction is that, for reasons of logical consistency, the higher-order formulas must be typed. No new implementation issues arise when a simple, non-polymorphic, form of typing is used and we implicitly assume such a typing regimen below; the treatment of polymorphic typing is considered in detail in [11]. Another difference is that, for technical reasons, the vocabulary of the higher-order logic includes the symbol $\top$ to denote the tautologous proposition and this symbol is considered to be an acceptable goal. The final and most significant difference is that first-order terms are replaced by the terms of a (simply typed) lambda calculus.

The lambda terms used in a higher-order logic can generally contain arbitrary quantifiers and connectives in them. However, for reasons explained in [17], our higher-order logic does not permit the terms that it uses to contain the symbols $\supset$ and $\sim$. The terms that result from omitting these symbols are referred to as *positive* terms. A (positive) atomic formula is then a formula of the form $P(t_1, \ldots, t_n)$ where P is a predicate constant or variable and, for $1 \leq i \leq n$, t_i is a positive term. Such a formula is said to be *rigid* in the case that P is a constant and *flexible* otherwise. Using the symbol A_r to represent rigid atomic formulas and A to denote arbitrary atomic formulas, the higher-order versions of goals and program clauses are given by the following syntax rules:

$$G ::= \top \mid A \mid (G \wedge G) \mid (G \vee G) \mid (\exists x\, G) \mid (Ds \supset G) \mid (\forall x\, G),$$
$$Ds ::= D \mid (D \wedge Ds), \text{ and}$$
$$D ::= A_r \mid (G \supset A_r) \mid (\forall x\, D).$$

From an implementation perspective, the main new concern in conjunction with our higher-order language is that first-order unification must be replaced by a notion of unification that incorporates equality based on λ-conversion. The resulting unification problem is different in several respects from first-order unification. In particular, the problem is undecidable in general and most general unifiers might not exist even when there are unifiers for given terms. There is, nevertheless, a procedure that can be used to find unifiers for these terms whenever they exist [9]. This procedure can be factored into the repeated application of certain simple steps and can be amalgamated as such into the abstract interpreter described in Section 4. A similar amalgamation has been carried out in [22] relative to a higher-order version of the Horn clause language and has been used in [20] to describe a WAM-based implementation scheme for this language. At a level of detail, the main new implementation concerns in the context of this language are (a) devising a good representation for lambda terms, (b) including machinery for performing λ-conversion, (c) incorporating a mechanism that supports the explicit representation of sets of terms that have to be unified, and (d) handling the possibility of branching within unification. The implementation scheme described in [20] contains a treatment of all these aspects. The language of interest here, the one described by the D and G formulas above, results essentially from adding universal quantifiers and implications as scoping devices to the higher-order Horn clause language. The approach to implementing these scoping mechanisms that we have presented in this paper carries over readily to the higher-order context. No significant changes are necessary with regard to the treatment of implication goals. The treatment of universal quantifiers must take into account the fact that predicate and function symbols can also be quantified over in the higher-order language. Tags must therefore be associated with these symbols as well and these must be used in the course of unification. An implementation of a higher-order language must already countenance the fact that variables and constants can be of function and predicate type and so the association of tags can be carried out in a manner entirely consistent with that described in this paper. The use of tags can be described as a simple check for tag compatibility at the time of binding a variable even in the context of the higher-order language [18], and this check can be implemented in a fairly transparent fashion.

A problem that is not directly addressed by the considerations above is that of complex goals that are generated dynamically. In the higher-order context, it is possible for a program to contain a goal of the form $P(a)$ where P is a variable. Now, P might be instantiated in the course of computation so that this goal becomes one that, for instance, has a universal quantifier as its top-level logical symbol. This raises the question of what code should be produced for the goal $P(a)$ by the compilation process. Clearly, it is not possible to anticipate the run-time form of this goal and so a compiler cannot produce code that accords a direct treatment to this form. However, an indirect treatment that fits in well with our current implementation scheme can be provided. The essential idea is to replace the goal $P(a)$ by the goal $solve(P(a))$, where $solve$ is a predicate that is defined by the clauses (written in pseudo-Prolog syntax)

$$
\begin{aligned}
&solve(G_1 \wedge G_2) \;\texttt{:-}\; (solve(G_1) \wedge solve(G_2)),\\
&solve(G_1 \vee G_2) \;\texttt{:-}\; (solve(G_1) \vee solve(G_2)),\\
&solve(\exists x\, G) \;\texttt{:-}\; (\exists x\, solve(G)), \text{ and}\\
&solve(\forall x\, G) \;\texttt{:-}\; (\forall x\, solve(G)),
\end{aligned}
$$

and a "clause" for the atomic case that results in setting up argument registers and then calling the appropriate predicate. The clauses for *solve* will themselves be compiled, and this results in a partial compilation of the actual goal that is produced from $P(a)$ at run-time. Note that the clauses for *solve* do not include one for the case of a dynamically created implication goal. The reason for this is that such a goal will never be produced in the context of our higher-order language: implications are prohibited from appearing in (lambda) terms. This situation is fortunate since it is not clear that a clause that can be compiled by the methods described in this paper can be provided for *solve* for this case.

The higher-order theory of hereditary Harrop formulas accounts for most of the examples presented in Section 2. However, there is one example that lies outside this theory and this corresponds to the final definition of `rev`. We reproduce this definition below (once again using pseudo-Prolog syntax):

```
rev(L1,L2) :-
(∀rev_aux ((rev_aux([],L2) ∧
            (∀X ∀L1 ∀L3 (rev_aux([X|L1],L3) :- rev_aux(L1,[X|L3]))))
               ⊃ rev_aux(L1,[]))).
```

The body of the clause defining `rev` has the form $\forall$`rev_aux` $(F \supset G)$, where F is a formula that represents the "clauses"

```
rev_aux([],L2).
rev_aux([X|L1],L3) :- rev_aux(L1,[X|L3]).
```

Notice, however, that these formulas are not really program clauses according to the current definition: the symbol `rev_aux` being a predicate variable, the "heads" of these formulas, *i.e.*, the expressions that appear on the right of the implication (or to the left of `:-`) in them, are not rigid atomic formulas as is required by our definition of D formulas.

The stipulation that the heads of program clauses be rigid atomic formulas is motivated by programming considerations. A program clause is to be thought of as a (partial) definition of a procedure, the name of the procedure that it defines being the top-level predicate symbol of its head. Such an interpretation would obviously not be very meaningful if this predicate symbol is a variable. The requirement of rigidity rules out this possibility. However, the example under consideration shows that this requirement is stronger than what might be needed. Thus, even though `rev_aux` is a variable, it will be replaced by a constant *before* the clauses "defining" it are added to the program and these clauses will constitute a meaningful procedure definition subsequent to such a replacement. Understanding this situation and noting that there is a useful paradigm embodied in the definition of `rev` under scrutiny, it seems worthwhile to extend our language to permit such definitions. We do this by enlarging our class of goals to include formulas of the form $\forall x\, F$ not only when F is a goal but also when F has the property that replacing all free occurrences of x in it by a constant c produces a goal. We intend, of course, that this acceptability condition for universally quantified goals be applied recursively. This intention can be embodied in a recursive definition, as is done in [6]. We do not do provide such a definition here, hoping that the intuitive content of the proposed enrichment is clear. In particular, it should be apparent that the definition of `rev` that is of interest is a bona fide program clause in the extended sense just described.

We consider now the additions needed for implementing our language under this extended definition of goals. An important requirement from this perspective is that of a means for establishing the identity of predicate constants, especially of those predicate constants that are introduced by the processing of universal quantifiers; such a scheme will be needed, for instance, in determining access to the clauses in the program. As we have already noted, every predicate constant will be assigned a tag under the present implementation scheme. We may, thus, think of an extended name for a predicate constant that is given by attaching the tag for the constant to it original name. The tag for "global" constants, *i.e.*, for constants like `rev` in the clause that appears earlier in this section, will be uniformly 1 and, hence, will not add much new information to the name. The tag value will, on the other hand, be a distinguishing characteristic of each predicate constant that is introduced in the course of processing a universal quantifier and that is available in a given context. In fact, the original name that is chosen for these constants may be ignored or considered to be a dummy one like *nil*, and the tag alone may be used in settling questions of identity.

In order to make the proposed naming scheme work, it is necessary to ensure that the tags associated with predicate constants are available wherever their names are needed. This is obviously the case for all global predicate constants. For a predicate constant that results from instantiating a universal quantifier, this issue needs to be considered relative to the variable occurrence that the constant replaces. When this variable occurrence is within an argument of an atomic formula, the machinery already in place ensures the availability of the tag information at the relevant time. In particular, the variable whose occurrence is being considered will be categorized as temporary or permanent and a binding and an associated tag will be determined for it by the processing of the relevant quantifier and, if the variable occurrence that is of interest is embedded within a clause in the antecedent of an implication goal, transmitted to the point of need by the execution of appropriate `initialize` instructions. In the case where the quantified variable occurs as the head of a goal that is to be invoked, the same considerations ensure that the tag value of the constant that replaces it will be known prior to the invocation of the goal. The only remaining case is that when the variable occurrence constitutes the "name" of a predicate defined by a clause in the antecedent of an implication goal. Some changes must be made to existing machinery in order to ensure that tagging information can be used in the desired manner in this case. To see this, let us return to the definition of `rev`. When the implication goal in its body is processed, such as in evaluating the query `rev([1,2,3],L)`, an implication point record will be created. This implication point record must provide access to the code for a procedure identified by the "constant" introduced for `rev_aux`. The main component of the name of this constant is, of course, its tag. However, this tag is known only at run-time. Hence, it cannot be included directly in the statically generated code that is associated with the implication point record and used in determining if the procedure being sought is the one defined by the clauses whose addition the record corresponds to.

The necessary tag information is, nevertheless, available at the time the implication point record is created and its use can be accommodated by making some changes to the compilation of implication goals. In particular, we think of the name of a predicate "constant" that is defined by clauses appearing in the antecedent of an implication goal as being given by a name and an offset. This offset is not used in the case of a global predicate constant, and the code for locating defining clauses for such a constant retains the shape described earlier. On the other hand, if the constant is one that is introduced by processing a universal quantifier, then the name component, which we will consider to be *nil*, becomes irrelevant, and the offset indicates the location in the environment record that was current at the time the implication point record was created where the binding for the quantified variable is stored and from where the tag may be obtained. When an attempt is made to solve an atomic goal or to fill in the vector *nc* in an implication point record, it may be necessary to locate code for predicate constants that are introduced by processing universal quantifiers. This task is carried out relative to an implication point record by comparing the tag associated with the constant and the tags obtained by using the `E'` field of the record and the offset numbers for the "hidden" predicates that are associated with the record.[10]

We consider now the compilation of atomic goals in conjunction with the scheme outlined above. Atomic goals whose predicate names are visible at the outermost level are compiled as before by using the `call` and `execute` instructions. The compilation of an atomic goal whose name is hidden by an enclosing universal quantifier requires the use of one of the instructions

```
call_value Vi,n
execute_value Vi
```

where `Vi` is a temporary or permanent variable. These instructions differ from the `call` and `execute` instructions only in the way they determine the location of the code to be invoked: this is done by determining the tag value associated with the constant that `Vi` is bound to and, assuming that this value is t, then searching from the most recent implication point record for code named by $\langle nil, t\rangle$.

The code that would be produced for `rev` using the ideas presented in this section is shown below. We assume in this code that `t1` is a pointer to the table created for the implication goal that appears in the body of the clause defining `rev`.

```
rev:      allocate 2             % rev_aux, L2 are permanent variables
          get_variable Y2,A2
          incr_universe          % ( ∀
          set_univ_tag Y1        %     rev_aux
          push_impl_point t1,2   %       add rev_aux code
          put_constant [],A2
          call_value Y1,2        %        call rev_aux
          pop_impl_point         %       restore earlier program
          decr_universe          % )
          deallocate
          proceed
```

The code that would be generated for the clauses in the antecedent of the implication goal that appears in the body of the definition of `rev` is shown below. The label $\langle$`nil,1`$\rangle$ is used here to indicate that this code is indexed by a predicate constant whose name component is *nil* and whose offset is 1.

```
⟨nil,1⟩:  try_me_else C1
          initialize X3,2        % X3 = L2
          get_constant [],A1
          get_value X3,A2        % unify L2 and second argument
          proceed

C1:       trust_me_else fail
```

[10]We have only presented a schematic solution to the problem here. In an actual implementation, the tags for hidden predicates may be precomputed at the creation of the implication point record and stored in it. Alternatively, this computation may be carried out when first needed and stored for subsequent use.

```
initialize X3,1       % X3 = rev_aux
get_list A1
unify_variable X4
unify_variable A1
get_variable X5,A2
put_list A2
set_value X4
set_local_value X5
execute_value X3
```

This code should be compared with the code shown for `rev_aux` in Section 7. The scoping effect of the universal quantifier warrants the conclusion that the second clause for `rev_aux` is the last one that can be used for solving it and, consequently, that the choice point record can be discarded prior to using it.

The scoping effect of the universal quantifier actually permits further improvements to be made to the code shown above for `rev` and `rev_aux`. First, the location of code that must be used in solving the consequent of the implication goal in the body of the clause for `rev` can be determined statically to be that which is labelled with $\langle$`nil,1`$\rangle$. The `call_value` instruction that appears in the code for `rev` can, therefore, be replaced by a direct call reminiscent of the WAM. A similar observation applies to the body of the second clause defining `rev_aux`, permitting the `execute_value` instruction appearing in the code for this predicate to be replaced by an `execute` instruction like that of the WAM. A further observation is that the code for `rev_aux` can be invoked from only these two places and so the implication point record that would be created in the course of solving the implication goal in the body of `rev` will not be needed for the purpose of accessing this code. As already noted, the two clauses for `rev_aux` could not be extending a previously existing definition for this predicate. Thus, the only purpose for the mentioned implication point record is that it maintains a binding for the variable `L2` that is free in the first clause for `rev_aux`. If an alternative means is provided for remembering this binding, the creation and removal of the implication point record can also be dispensed with.

Observations such as those above can lead to significant efficiency improvements in the code that is produced. It seems worthwhile, therefore, to develop methods of static analysis that allow such observations to be made. Note, however, that, even after such a static analysis, a complete treatment of the current language will still require the issues examined in this section to be dealt with. In particular, there are situations in which definitions of predicates whose names are hidden by universal quantifiers actually change in the course of computation. The location of the code for such predicates can therefore not always be determined statically and some mechanism must be provided both for dynamically extending existing definitions and for identifying the relevant code at run-time. To understand these comments, let us consider the following goal (presented, again, in pseudo-Prolog syntax) in which `a` and `b` are constants and `r` and `s` are predicates that are defined by clauses in the (implicit) global program:

```
∀p ∀q (((∀X (p(X) :- q(X))) ∧
       (∀Y (q(Y) :- r(Y))))
              ⊃ (p(a) ∧ ((∀Z (q(Z) :- s(Z))) ⊃ p(b)))).
```

Assume that the constants introduced in processing the two outermost universal quantifiers are named `p` and `q` respectively. Then, solving the given goal eventually requires the two goals `p(a)` and `p(b)` to be solved. The definition of `p` in both cases is given by the clause

```
∀X(p(X) :- q(X)).
```

Notice, however, that the definition of `q` is different in the two cases. When `p(a)` is to be solved, `q` will be defined by the sole clause

```
∀Y(q(Y) :- r(Y)).
```

Prior to solving `p(b)`, this definition will be extended by the addition of the clause

```
∀Z(q(Z) :- r(Z)).
```

Thus, despite the universal quantification over `q`, the occurrence of `q` in the clause defining `p` cannot be compiled into a direct call. Some mechanism that supports the extension of definitions even for such predicates and that facilitates the resolution of identity questions pertaining to them therefore appears necessary.

9 CONCLUSION

In this paper we have considered an enrichment to logic programming that is based on allowing implications and universal quantifiers to appear in goals. We have argued that the inclusion of these symbols leads to several novel features at a programming level, including a means for giving names and programs a scope. We have then discussed the implementation problems that arise from the addition of these symbols. These problems are of three broad kinds:

(i) The possibility for existential and universal quantifiers to occur in mixed order in goals requires a careful treatment of unification. In particular, instantiations for variables must respect the order in which quantifiers appear.

(ii) Programs may change in the course of computation by the addition or removal of clauses and a mechanism is needed for implementing these changes in an incremental fashion. Furthermore, backtracking may cause a return to a previously existing program context and so it should be possible to resurrect such contexts quickly.

(iii) A method is needed for representing program clauses that permits compilation and the sharing of compiled code even though the exact form of these clauses may be dynamically determined.

We have presented solutions to these problems. Our solution to the first problem is based on an association of tags with constants and variables and the use of these tags to ensure that variable bindings determined during unification respect the necessary constraints. With regard to the second problem, we have proposed a new kind of record called an implication point record that represents the creation of a new program by the addition of a certain set of clauses to a previously existing program. Implication point records are to be maintained on the local stack and each of them will be retained as long as there is a possibility to return to the program context that it represents. The resurrection of an earlier program context can therefore be achieved simply by switching to the appropriate implication point record. Finally, as a solution to the last problem, we have described a closure-based representation of program clauses. This representation separates each clause into a fixed part that can be compiled (and shared) and an environment that records the part that is dynamically determined. A feature of the solutions that we have developed to the problems described above is that they can all be easily integrated into the structure of the WAM. We have described this integration and have discussed the issue of compiling programs in our extended language into instructions that will run on the resulting machine.

Although the focus in this paper has been on a first-order language, the ultimate objective of our work is to provide an implementation of a polymorphically typed, higher-order version of this language. The ideas that we have developed here for implementing the scoping mechanisms are, as we have indicated, not dependent on whether these mechanisms are being added to a first-order or a higher-order language. There are, however, substantial additional issues that have to be considered in implementing the desired form of typing and in realizing higher-order aspects. We have considered these issues in detail elsewhere [11, 19, 20, 23]. We have also combined the ideas that we have developed for implementing these aspects with those in this paper to produce an abstract machine for the overall language. The development of an emulator for this machine and of a compiler for translating programs in the extended language into instructions that will run on this machine is currently being undertaken.

10 ACKNOWLEDGEMENTS

This paper has benefitted from the suggestions for improvements made by its referees. Some of the ideas in this paper were previously presented in [10] and we are grateful to Evelina Lamma, Dale Miller and Paola Mello for their comments on this presentation.

REFERENCES

[1] Hassan Aït-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction.* MIT Press, 1991.

[2] Pascal Brisset and Olivier Ridoux. Naive reverse can be linear. In Koichi Furukawa, editor, *Eighth International Logic Programming Conference*, pages 857–870, Paris, France, June 1991. MIT Press.

[3] W. F. Clocksin and C. S. Mellish. *Programming in Prolog.* Springer-Verlag, 1984.

[4] Conal Elliott and Frank Pfenning. A semi-functional implementation of a higher-order logic programming language. In Peter Lee, editor, *Topics in Advanced Language Implementation*, pages 289–325. MIT Press, 1991.

[5] Amy Felty. *Specifying and Implementing Theorem Provers in a Higher-Order Logic Programming Language.* PhD thesis, University of Pennsylvania, August 1989.

[6] Elsa L. Gunter. Extensions to logic programming motivated by the construction of a generic theorem prover. In Peter Schroeder-Heister, editor, *Extensions of Logic Programming: International Workshop, Tübingen FRG, December 1989*, pages 223–244. Springer-Verlag, 1991. Volume 475 of *Lecture Notes in Artificial Intelligence.*

[7] John J. Hannan. *Investigating a Proof-Theoretic Meta-Language for Functional Programs.* PhD thesis, University of Pennsylvania, August 1990.

[8] J. Roger Hindley and Jonathan P. Seldin. *Introduction to Combinatory Logic and Lambda Calculus.* Cambridge University Press, 1986.

[9] Gérard Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1:27–57, 1975.

[10] Bharat Jayaraman and Gopalan Nadathur. Implementation techniques for scoping constructs in logic programming. In Koichi Furukawa, editor, *Eighth International Logic Programming Conference*, pages 871–886, Paris, France, June 1991. MIT Press.

[11] Keehang Kwon, Gopalan Nadathur, and Debra Sue Wilson. Implementing polymorphic typing in a logic programming language. *Computer Languages*, 20(1):25–42, 1994.

[12] Evelina Lamma, Paola Mello, and Antonio Natali. The design of an abstract machine for efficient implementation of contexts in logic programming. In G. Levi and M. Martelli, editors, *Sixth International Logic Programming Conference*, pages 303–317, Lisbon, Portugal, June 1989. MIT Press.

[13] Dale Miller. Hereditary Harrop formulas and logic programming. In *Proceedings of the VIII International Congress of Logic, Methodology, and Philosophy of Science*, pages 153–156, Moscow, August 1987.

[14] Dale Miller. Lexical scoping as universal quantification. In G. Levi and M. Martelli, editors, *Sixth International Logic Programming Conference*, pages 268–283, Lisbon, Portugal, June 1989. MIT Press.

[15] Dale Miller. A logical analysis of modules in logic programming. *Journal of Logic Programming*, 6:79–108, 1989.

[16] Dale Miller and Gopalan Nadathur. A logic programming approach to manipulating formulas and programs. In Seif Haridi, editor, *IEEE Symposium on Logic Programming*, pages 379–388, San Francisco, September 1987.

[17] Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

[18] Gopalan Nadathur. A proof procedure for the logic of hereditary Harrop formulas. *Journal of Automated Reasoning*, 11(1):115–145, August 1993.

[19] Gopalan Nadathur. A notation for lambda terms II: Refinements and applications. Technical Report CS-1994-01, Department of Computer Science, Duke University, January 1994.

[20] Gopalan Nadathur, Bharat Jayaraman, and Debra Sue Wilson. Implementation considerations for higher-order features in logic programming. Technical Report CS-1993-16, Department of Computer Science, Duke University, June 1993.

[21] Gopalan Nadathur and Dale Miller. An overview of λProlog. In Kenneth A. Bowen and Robert A. Kowalski, editors, *Fifth International Logic Programming Conference*, pages 810–827, Seattle, Washington, August 1988. MIT Press.

[22] Gopalan Nadathur and Dale Miller. Higher-order Horn clauses. *Journal of the ACM*, 37(4):777–814, October 1990.

[23] Gopalan Nadathur and Debra Sue Wilson. A notation for lambda terms I: A generalization of environments. Technical Report CS-1994-03, Department of Computer Science, Duke University, January 1994.

[24] Remo Pareschi and Dale Miller. Extending definite clause grammars with scoping constructs. In David H. D. Warren and Peter Szeredi, editors, *1990 International Conference in Logic Programming*, pages 373–389. MIT Press, June 1990.

[25] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Proceedings of the ACM-SIGPLAN Conference on Programming Language Design and Implementation*, pages 199–208. ACM Press, June 1988.

[26] D.H.D. Warren. An abstract Prolog instruction set. Technical Note 309, SRI International, October 1983.