TOWARDS A WAM MODEL FOR 
\lprolog

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

BHARAT JAYARAMAN  
Department of Computer Science  
State University of New York at Buffalo  
Buffalo, NY 14260  

Abstract: Issues concerning the implementation of the logic programming language \lprolog are investigated. This language extends Prolog by providing for higher-order notions and incorporating a richer set of search primitives. Techniques must therefore be devised for implementing these two new aspects. Extensions to the Warren Abstract Machine (WAM) are proposed in order to deal with the several significant new problems that arise in conjunction with the former aspect. A representation for \l-terms is outlined that facilitates the examination of their internal structure while simultaneously permitting reductions to be performed easily. Mechanisms are discussed for implementing higher-order unification — an operation that is much more complex than first-order unification — within the backtracking paradigm of Prolog. Instructions for creating \l-terms and invoking operations within unification are also provided. The nature of compiled code is illustrated through examples, and it is argued that the enhancements to the WAM preserve its behavior over first-order programs. The ideas presented in this paper are the basis of an implementation effort that is currently underway.

1. Introduction

This paper considers implementation issues concerning \lprolog [8], a logic programming language that extends Prolog by supporting higher-order programming, incorporating \l-terms as data structures and providing mechanisms for defining modules and abstract data types. Two interpreters are currently in existence for this language, one of these being the experimental system LP2.7 [6] and the other being the system eLP under development at Carnegie-Mellon University.
Experiments conducted with LP2.7 have indicated the usefulness of \( \lambda \)Prolog, particularly of its higher-order aspects, as an implementation vehicle in areas as diverse as natural language understanding, proof systems and program transformation systems (see [8] for references). There is, thus, considerable value for an efficient implementation of \( \lambda \)Prolog. Towards providing such an implementation, we examine the extension of the Warren Abstract Machine (WAM) [12] to this language and also consider issues of compilation in this paper.

The basis for \( \lambda \)Prolog lies in a class of formulas called higher-order hereditary Harrop or \( \text{hohh} \) formulas [7]. At a conceptual level, \( \text{hohh} \) formulas extend Horn clauses in two orthogonal directions. First, they embed higher-order notions and, second, they provide primitives for specifying two new search operations. Thus, in adapting the WAM to \( \lambda \)Prolog, mechanisms for implementing these two new aspects have to be described. Our focus here is on the implementation of the higher-order notions and we therefore restrict our attention to a subset of \( \text{hohh} \) formulas called higher-order Horn clauses [9] that incorporate only this additional feature. Even when restricted to this context, there are several significant new problems to be dealt with. A representation for \( \lambda \)-terms has to be devised that facilitates the examination of their internal structure while simultaneously permitting \( \beta \)-reductions on such terms to be performed easily. Similarly, mechanisms for implementing higher-order unification need to be provided. In implementing this operation, some problems that do not arise in the first-order context have to be solved: for instance, the sets of terms that need to be unified must be maintained explicitly and the possibility of branching within unification must be handled. Finally, the various implementation techniques devised must pay heed to the fact that they are to be embedded within the backtracking paradigm of Prolog.

The enhanced WAM that we describe in this paper incorporates a solution to each of the problems discussed above. The salient characteristics of this adaptation to the WAM are the following:

1. The de Bruijn representation [3] is used for \( \lambda \)-terms. Also \( \beta \)-reduction avoids copying the argument of an application by using

\[ \text{It is sometimes mistakenly believed that these problems have been addressed by Warren in [11]. Techniques described in [11] deal only with extremely restricted forms of predicate quantification. The first careful analysis of higher-order notions in logic programming appears to be provided in [5] and implementation techniques for the resulting language have not been discussed to date.} \]
environments for representing variable bindings. Finally all λ-terms and their environments are allocated on the heap, along with Prolog's first-order terms, to facilitate storage reclamation during backtracking.

2. The branching search in higher-order unification is managed by creating branch points — which are similar to WAM's choice points — on the local stack. The disagreement sets generated during higher-order unification are managed using two threaded-stacks, which facilitate both easy deletion of inapplicable constraints and re-instatement of earlier constraints during backtracking.

3. A few new WAM instructions are introduced for building λ-terms. The get and unify instructions are modified so that they work both on first-order and higher-order terms. Two new instructions are introduced exclusively for higher-order unification.

An important property of our abstract machine is that it seems to compare favorably with the WAM over the class Prolog programs, i.e. over first-order Horn clauses. Although a final judgement on this issue must await an actual implementation, a strong argument may already be made to support this claim: Prolog programs would not cause the invocation of any of the additional machinery provided for implementing higher-order unification, even though no global analysis is made to distinguish cases when the arguments to goals are higher-order terms.

The remainder of this paper is structured as follows. In the next two sections we describe higher-order Horn clauses and outline an abstract interpreter for a logic programming language based on them. This description summarizes those found in, e.g., [9]. We then describe an enhanced WAM model supporting higher-order unification, and conclude with examples of compiled code for sample programs.

2. Higher-Order Horn Clauses

In defining higher-order Horn clauses, use is made of the simply typed λ-terms of [2]. The types in these terms are those obtained from a set S of sorts by using a set C of type constructors each of specified arity and the function type constructor ↦. In particular, (i) each member of S is a type, (ii) (c α₁ ... αₙ) is a type for each c ∈ C of arity n and each α₁,..., αₙ that are types, and (iii) (α ↦ β) is a type for each α, β that are types. We shall assume that S contains the sorts int and o for integers and booleans respectively, and that C contains the unary list type constructor list. A type is atomic if
it is obtained by virtue of (i) or (ii) and a function type otherwise. Assuming that \( \rightarrow \) is right associative, a function type may be written in the form \( \alpha_1 \rightarrow \ldots \rightarrow \alpha_n \rightarrow \beta \) where \( \beta \) is an atomic type; \( \beta \) is then said to be its target type and \( \alpha_1, \ldots, \alpha_n \) are its argument types. This notation and terminology is extended to atomic types by permitting the argument types to be an empty sequence.

The terms are obtained from denumerable sets of variables and constants whose elements have specified types as follows: (i) each constant and variable of type \( \alpha \) is a term of type \( \alpha \), (ii) if \( x \) is a variable of type \( \alpha \) and \( t \) is a term of type \( \beta \), then \( (\lambda x \ t) \) is a term of type \( (\alpha \rightarrow \beta) \) and is called an abstraction that binds \( x \) and has body \( t \), and (iii) if \( t_1 \) and \( t_2 \) are terms of type \( (\alpha \rightarrow \beta) \) and \( \alpha \) respectively, then \( (t_1 \ t_2) \) is a term of type \( \beta \) and is called an application of \( t_1 \) to \( t_2 \).

The constants are partitioned into parameters and the logical constants \( T \) of type \( o \), \( \Lambda \), \( \forall \) and \( \exists \) of type \( o \rightarrow o \rightarrow o \) and, for each \( \alpha \), \( \exists_\alpha \) and \( \forall_\alpha \) of type \( (\alpha \rightarrow o) \rightarrow o \). In writing terms, the type subscripts on \( \exists \) and \( \forall \) shall often be dropped and expressions of the form \( \exists (\lambda x \ B) \) and \( \forall (\lambda x \ B) \) shall be abbreviated by \( \exists x B \) and \( \forall x B \) respectively. Also \( \wedge \), \( \vee \) and \( \supset \) shall be written in the usual infix manner and parentheses shall be minimized by assuming abstraction and application are, respectively, right and left associative.

Let \( A \) denote a boolean term of the form \( (P \ t_1 \ldots \ t_n) \) where \( P \) is a variable or parameter and the \( t_i \)'s are terms not containing \( \supset \) or \( \forall \) and let \( A_r \) denote such a term in the case that \( P \) is a parameter; such terms are called positive and rigid positive atoms respectively. Goal formulas, denoted by \( G \), are then boolean terms whose structure is given as follows:

\[
G := A \mid T \mid G_1 \wedge G_2 \mid G_1 \vee G_2 \mid \exists x G.
\]

Finally a higher-order Horn clause is the (implicit) universal closure of terms of the form \( A_r \) or \( G \supset A_r \).

A set of higher-order Horn clauses constitutes a program and a goal formula corresponds to a query. A computation then consists of proving an instance of the query from the program and returning the corresponding substitution as an answer. For example, if \( \text{mapfun} \) is of type \( ((\text{list} \ \text{int}) \rightarrow (\text{int} \rightarrow \text{int}) \rightarrow (\text{list} \ \text{int}) \rightarrow o) \), the clauses

\[
\begin{align*}
\text{(mapfun } \text{[]} \ F \text{ } \text{[]}\text{).} \\
\text{(mapfun } \text{[(x)} \text{1]} \ F \text{ } \text{[(F \ x)} \text{2]} \text{) } \vdash \text{ (mapfun } L_1 \ F \text{ } L_2 \text{).}
\end{align*}
\]

\[\dagger\] In reality \( \lambda \text{Prolog} \) provides for polymorphic types through the use of type variables and also permits such a type to be inferred for \( \text{mapfun} \). We do not deal with these aspects in this paper.
describe a program; implication is written here in the more suggestive manner of Prolog and use is made of Prolog's list notation and of its convention for distinguishing constants from variables. Examples of queries are the formula \((\text{mapfun}\ (\lambda x (g\ 1\ x))\ [1,2] \ L)\) and the formula \((\text{mapfun}\ F\ [1,2]\ [(g\ 1\ 1),(g\ 1\ 2)])\). Given the above program, answers to these queries are the values \([(g\ 1\ 1),(g\ 1\ 2)]\) for \(L\) and \(\lambda x (g\ 1\ x)\) for \(F\), respectively. It is of particular interest to note that computing the latter answer requires an essential use of higher-order unification.

3. An Abstract Interpreter

SLD-derivations [1] can be extended to higher-order Horn clauses [9]. The new aspect that needs to be considered is higher-order unification. Given a disagreement set, i.e. a finite set \(\{(t_1, s_1), \ldots, (t_n, s_n)\}\) of pairs of terms of the same type, this problem involves finding a substitution \(\sigma\) such that \(\sigma(t_i)\) is \(\lambda\)-convertible† to \(\sigma(s_i)\) for \(1 \leq i \leq n\). The task of finding such a substitution, called a unifier for the given set, is fairly complex and may involve a branching search. Unlike first-order unification, it can therefore not be treated as an atomic operation within SLD-derivations. However, it is possible to factor the search for unifiers into atomic operations [4], and these can be integrated into SLD-derivations to yield the notion of a \(P\)-derivation that is outlined below.

Each term can be represented in head-normal form, i.e. in the form

\[
\lambda x_1 \ldots \lambda x_n (A\ t_1 \ldots\ t_m)
\]

where \(A\) is a constant or variable of type \(\alpha_1 \rightarrow \cdots \rightarrow \alpha_m \rightarrow \beta\). Given such a representation, \(A\) is called the head of the term, and the term is said to be rigid if \(A\) is a constant or is an element of \(\{x_1, \ldots, x_n\}\), and flexible otherwise. Given two rigid terms that are of the same type and hence can be written as \(\lambda x_1 \ldots \lambda x_n (A_1\ s_1 \ldots\ s_i)\) and \(\lambda x_1 \ldots \lambda x_n (A_2\ r_1 \ldots\ r_j)\) respectively, it can be seen that these terms are unifiable only if \(A_1\) and \(A_2\) are identical. Further, they then have the same set of unifiers as the disagreement set

\[
\{(\lambda x_1 \ldots \lambda x_n s_1, \lambda x_1 \ldots \lambda x_n r_1), \ldots, (\lambda x_1 \ldots \lambda x_n s_i, \lambda x_1 \ldots \lambda x_n r_i)\}.
\]

Given an arbitrary disagreement set, we may thus either conclude that it has no unifiers or reduce it to another disagreement set with

† The rules of \(\alpha\-, \beta\-\) and \(\eta\)-conversion are assumed to comprise the \(\lambda\)-conversion rules here.
the same unifiers and in which each pair has at least one flexible term; the preceding observation provides the basis for performing a walk over the "first-order" structure of terms to produce this effect. The function SIMPL is assumed to carry out this process of reduction below, returning either a disagreement set or the marker F when the impossibility of unification is detected.

A unifier can readily be provided for a set consisting only of "flexible-flexible" pairs. Thus the only other aspect is that of reducing the difference between a flexible and a rigid term, i.e., a "flexible-rigid" pair. Two kinds of elementary substitutions may be employed for this purpose. The first makes the head of the flexible term "imitate" that of the rigid term, and the second "projects" it onto one of the arguments in the hope that the head of the resulting term may be made identical to the rigid one. In particular, let $F_1 = \lambda x_1 \ldots \lambda x_n (f_1 t_1 \ldots t_k)$ and $F_2 = \lambda x_1 \ldots \lambda x_n (c s_1 \ldots s_j)$ be the flexible and rigid terms in head normal form. Further, let the type of $f$ be $\alpha_1 \rightarrow \cdots \rightarrow \alpha_k \rightarrow \beta$. Then

(i) $IMIT(F_1, F_2)$, the imitation substitution, is defined only when $c$ is a variable and is the following:

$$\{(f, \lambda w_1 \ldots \lambda w_k (c \ (h_1 w_1 \ldots w_k) \ldots \ (h_j w_1 \ldots w_k)))\}$$

(ii) for $1 \leq i \leq k$, $PROJ_i(F_1, F_2)$, the $i^{th}$ projection substitution, is defined only when $\alpha_i$ is of the form $\beta_1 \rightarrow \cdots \rightarrow \beta_i \rightarrow \beta$ and is the following:

$$\{(f, \lambda w_1 \ldots \lambda w_k (w_i \ (h_1 w_1 \ldots w_k) \ldots \ (h_l w_1 \ldots w_k)))\}.$$  

The $hs$ above are assumed to be new variables of the appropriate types. Observe that these substitutions are determined entirely by the heads of the flexible and rigid terms. We assume below that MATCH is a function on flexible-rigid disagreement pairs that produces the (finite) set of imitation and projection substitutions.

Let $P$ be a set of higher-order Horn clauses. Further, let $G$, $D$ and $\theta$ be symbols for sets of goal formulas, disagreement sets and substitutions respectively. Then the tuple $(G_2, D_2, \theta_2)$ is said to be $P$-derivable from the tuple $(G_1, D_1, \theta_1)$ if $D_1 \neq \emptyset$ and, in addition, one of the following situations holds:

1. (Goal reduction step) $\theta_2 = \emptyset$, $D_2 = D_1$, and for some $G \in G_1$ it is the case that
   
   (a) $G$ is T and $G_2 = G_1 - \{G\}$, or
   
   (b) $G$ is $G_1 \land G_2$ and $G_2 = (G_1 - \{G\}) \cup \{G_1, G_2\}$, or
   
   (c) $G$ is $G_1 \lor G_2$ and, for $i = 1$ or $i = 2$, $G_2 = (G_1 - \{G\}) \cup \{G_i\}$, or
(d) G is \( \exists x P \) and \( \mathcal{G}_2 = (\mathcal{G}_1 - \{G\}) \cup \{(\lambda x \, P \, y)\} \) where y is a new variable.

(2) (Solving flexible goals) G ∈ \( \mathcal{G}_1 \) has the variable y of type \( \alpha_1 \to \cdots \to \alpha_n \to \beta \) as its head, and \( \theta_2 = \{(y, \lambda x_1 \cdots \lambda x_n \, T)\} \), \( \mathcal{G}_2 = \theta_2(\mathcal{G}_1 - \{G\}) \) and \( \mathcal{D}_2 = \text{SIMPL}(\theta_2(\mathcal{D}_1)) \).

(3) (Backchaining step) \( \theta_2 = \emptyset \) and, for some rigid atom G ∈ \( \mathcal{G}_1 \) and a variant \( G' \supset A \) of a clause in \( \mathcal{P} \) with new free variables, \( \mathcal{G}_2 = (\mathcal{G}_1 - \{G\}) \cup \{G'\} \), and \( \mathcal{D}_2 = \text{SIMPL}(\mathcal{D}_1 \cup \{(G, A)\}) \).

(4) (Unification step) For some flexible-rigid pair \( \chi \in \mathcal{D}_1 \), either MATCH(\( \chi \)) = \emptyset and \( \mathcal{D}_2 = \mathcal{F} \), or \( \theta_2 \in \text{MATCH}(\chi) \) and \( \mathcal{G}_2 = \theta_2(\mathcal{G}_1) \) and \( \mathcal{D}_2 = \text{SIMPL}(\theta_2(\mathcal{D}_1)) \).

A sequence of the form \( (\mathcal{G}_i, \mathcal{D}_i, \theta_i)_{1 \leq i \leq n} \) is a \( \mathcal{P} \)-derivation sequence for a goal formula G if (i) \( \mathcal{G}_1 = \{G\} \) and (ii) for 1 ≤ j ≤ n, the \( (j + 1)^{\text{th}} \) tuple is \( \mathcal{P} \)-derivable from the \( j^{\text{th}} \) tuple. Such a sequence evidently terminates if \( \mathcal{D}_n = \mathcal{F} \) or \( \mathcal{G}_n = \emptyset \) and \( \mathcal{D}_n \) is either empty of contains only flexible-flexible pairs. In the latter case, we say that the sequence is a \( \mathcal{P} \)-derivation for G. Such a sequence in fact constitutes a proof for G from \( \mathcal{P} \), and an answer substitution may be obtained by composing \( \theta_n \circ \cdots \circ \theta_1 \) with any unifier for \( \mathcal{D}_n \). An abstract interpreter for our language may thus be thought of as a procedure that, given a program \( \mathcal{P} \), attempts to construct a \( \mathcal{P} \)-derivation for goal formulas.

4. An Enhanced WAM Model

The abstract interpreter described in Section 3 has to exercise several choices in trying to construct a \( \mathcal{P} \)-derivation. The manner in which some of these choices are made impinges critically on the completeness of the interpreter. It is necessary, for instance, to delay the solving of flexible goals until none of the other steps can be applied. Similarly, the choices of (a) the disjunct in a goal reduction step involving a disjunctive goal, (b) the clause in a backchaining step, and (c) the substitution in a unification step are critical. In the enhanced WAM model described below, we assume these choices are exercised in a manner compatible with Prolog implementations: goals are never reordered and a depth-first search is conducted. In addition we assume that a unification step is performed whenever possible, i.e. unification problems are always "solved" first. The search for unifiers, of course, proceeds in a depth-first fashion. There is, therefore, much that can be retained from the WAM in dealing with higher-order Horn clauses. There are, however, three issues that require extensions to this model to be considered:

(i) the need to represent \( \lambda \)-terms in a manner that facilitates the
examination of their internal structure while permitting the operation of \( \beta \)-reduction on such terms to be performed easily;

(ii) the need to accommodate branching that arises due to multiple substitutions provided by MATCH;

(iii) the need to maintain disagreement sets and, in particular, to carry “flexible-flexible” disagreement sets over a backchaining step.

We show how these issues can be treated within an extension to the WAM. We assume the reader has some familiarity with the basic WAM issues, i.e., the structure-copying representation of terms, and the use of the local stack, heap and trail. Our description concentrates on the new features needed in dealing with the problems mentioned above.

**Representation of \( \lambda \)-terms and \( \beta \)-reduction**

Our representation of \( \lambda \)-terms is an adaptation of the namefree scheme devised by de Bruijn [3]. Within this scheme, a bound variable occurrence is represented by an index that counts the number of abstractions from the occurrence up to the abstraction binding it. One difference between our proposed scheme and that of de Bruijn’s is that variables that appear free in clauses and goals are represented by direct pointers in our scheme rather than by indices obtained from an implicit listing of free variables. An advantage of the namefree representation is that there is never any need for considering alphabetic variants of terms. Thus, when the head-normal forms of two rigid terms are checked for compatibility, an identity test suffices. Figure 1 illustrates the representation of a typical term. The notation \( i \) is used to refer to a variable that is bound by an abstraction occurring \( i \) “levels” above, and \( @ \) is used to represent an application. Although \( @ \) is treated as a binary operator, the left associativity of application permits us to collapse a sequence of \( @ \)s into one in which the second argument is a list of terms. This fact is made use of implicitly later in the paper, yielding, in particular, a direct correspondence between the usual representation of first-order terms and the one accorded to them within our scheme.

In obtaining the head-normal form of a \( \lambda \)-term, it may be necessary to perform \( \beta \)-reductions. Performing a \( \beta \)-reduction involves making a substitution within the body of an abstraction for each occurrence of the bound variable of the abstraction. There are two criteria that drive the choice of machinery for performing such a substitution. First, it is desirable to compute the effects of the substitution only
on demand, thus avoiding work that may be made unnecessary by backtracking. Second, the replication of substitution terms should be avoided and an attempt should be made to share structure wherever possible. Both these purposes are achieved by recording the substitution necessitated by a $\beta$-reduction in an "environment" and then interpreting the abstraction body within this environment. However, a naive use of the usual closure mechanism is precluded by the adoption of the de Bruijn representation and the need to examine terms embedded within abstractions. For instance, when a $\beta$-reduction is performed within a sequence of abstractions, it is necessary to record the fact that the indices for variables bound by the outer abstractions must be decremented. Similarly in "transferring" a binding into the body of an abstraction, it must be remembered that the indices for the free variables in the term to be substituted must be incremented.

In view of these additional constraints, the data structure that is actually used to represent a term consists of a triple, called a suspension in [10]. The first component of this triple is a pointer to a "skeletal" $\lambda$-term (which may in turn be represented by such a triple). The second component is an integer, called the level, that records the number of abstractions that the term being represented is to be in-

**Figure 1**: de Bruijn representation of $\lambda Z((\lambda W ((W a) (ZW))) (\lambda Y Y))$
interpreted as being embedded within. The last component is what is called a \(\lambda\)-environment; the term "\(\lambda\)-environment" is used to avoid confusion with the term environment record, used for clauses in the WAM model. The \(\lambda\)-environment contains an entry, in reverse order to that of its appearance, for each abstraction encountered in descending to the term under consideration. For abstractions that have now disappeared due to a \(\beta\)-reduction, the entry consists of a term repre-
sented in a similar fashion. For abstractions that persist, this consists of a “dummy” entry of the form \((dum, Lev, Env)\) whose chief purpose is to record the level at the point when the abstraction was encountered. Figure 2 illustrates the use of this data structure in the process of head-normalizing the \(\lambda\)-term of Figure 1. A detailed discussion of this data structure and the operations on it appear in [10] and are beyond the scope of this paper; suffice it to say that the representation encodes enough information to obtain the correct de Bruijn indices at the point desired while at the same time permitting substitutions to be performed lazily. A particular point to note is that the second and third components may be omitted when they are, respectively, 0 and empty. Thus, in the context of first-order terms, the representation is quite akin to that used in the conventional WAM.

We note that space for all \(\lambda\)-terms is allocated on the heap along with other Prolog terms, as this facilitates automatic space reclamation during backtracking. Space for \(\lambda\)-environments is also allocated on the heap. The \(\lambda\)-environment is represented as a linked list of bindings. We believe that such a representation should suffice because environments may in general not be long. If this turns out not to be true in practice, an array or “display” representation may be used.

**Disagreement Sets and Deferred Constraints**

The decidability of first-order unification as well as the existence of most general unifiers permits the task of unifying two first-order terms to be decomposed into that of (incrementally) unifying each of their subterms. This obviates the need for maintaining explicit representations of disagreement sets in this case. In the higher-order context, however, this does not appear to be possible: “flexible-flexible” disagreement sets need to be carried over clause invocations and it seems preferable to maintain and manipulate such sets even across a sequence of unification steps. In determining a representation for these sets, three factors seem to require consideration:

(i) incremental changes to an existing disagreement set should reuse as much of the original representation as possible (i.e. copying entire disagreement sets should be avoided);

(ii) it should be possible to rapidly re-instate earlier disagreement sets when backtracking over a choice point or branch point occurs;

(iii) it should be possible to quickly determine whether such a set has flexible-rigid disagreement pairs and to obtain a pair of this sort if it does.
To realize the first requirement, we use a linked stack representation for disagreement sets, i.e., entries in the stack are also linked together. To realize the second requirement, we actually use a doubly-linked list. This scheme works as follows. The "current" disagreement set is indicated by a live list that is obtained by threading the relevant pairs in the stack. A call to SIMPL — something that occurs right after a substitution is determined or within a backchaining step — may require some of the elements in this live list to be removed and new ones to be added. The former is done by splicing the elements out of the live list and putting pointers to them on a separate trail stack. The addition of new elements is effected by first placing them on the top of the stack and then adding them onto the end of the live list. When backtracking, the changes caused by SIMPL have to be undone. In particular, the dead elements have to be put back, the new live elements have to be removed and their space reclaimed.

To realize the third requirement, we implement the above scheme using two doubly-linked stacks, one for flexible-rigid pairs (called FR stack) and the other for flexible-flexible pairs (called FF stack). Likewise, we use two separate trail stacks called FR trail and FF trail respectively. New WAM global registers needed for the FR and FF stacks are the following: FR and FF which are pointers to the top of the FR and FF stacks; FR_LL and FF_LL which are pointers to the live lists that run through the FR and FF stacks; and FR_TR and FF_TR which are pointers to the top of the FR and FF trail;

A point to note is that our control strategy calls for the application of unification steps until the disagreement set is reduced either to the empty set or to a set consisting solely of flexible-flexible pairs. While sets of the latter sort are known to have unifiers, the search for their unifiers can be fairly unconstrained. Our implementation avoids this search by proceeding to a backchaining step when the FR_LL register points to an empty list. The remaining flexible-flexible pairs that in effect constitute deferred constraints are implicitly carried forward by the FF_LL register.

Branching Within Unification
The WAM model uses environment records and choice-point records on the local stack to keep track of the variables in a clause and backtracking information respectively. In addition to these two, we have a branch-point record, which is pushed on the local stack for each branch point (IMIT or PROJ) within MATCH. The details of branch-point records are shown in Figure 3, but the basic differences from a choice-point record are:
Figure 3: Enhanced WAM machine state

(a) the next-clause field of a choice-point record is replaced by a
Match-Status field, which (i) indicates whether IMIT or PROJ is in progress, and, in the case of PROJ, which particular projection is in progress, and (ii) has a pointer to the disagreement pair under consideration;

(b) space for the registers of the caller (in the choice-point record) is replaced by two fields: the FF-Status and the FR-Status fields, which record information about the state of FF and FR stacks respectively. Each of these status fields has the following three components: (i) a pointer to the top of the FF (resp. FR stack), (ii) a pointer to the top of the FF (resp. FR) trail and (iii) a pointer to the end of the live list.

Upon backtracking to a branch/choice point, two actions are taken with respect to the FR and FF stacks: (i) all pairs above the points indicated by the first components of the FR-Status and FF-Status are deleted from these two stacks respectively; (ii) the pairs pointed to by the addresses in the FF and FR trail stacks are linked back into the live lists of the FR and FF stacks respectively; and (iii) the end of the live list is restored.

Note that the λ-terms and variables introduced by IMIT and PROJ in defining a substitution are allocated space on the heap. This is an appropriate place for their allocation, because backtracking will automatically reclaim this storage. Also, with respect to backtracking, branch points and choice points can be treated alike, and hence they are linked together into a single list.

Figure 3 summarizes the state of an extended WAM machine for a λProlog program. The details in the figure should be self-explanatory, in view of earlier explanations.

5. Compilation

The philosophy underlying the WAM is to compile as much of the unification and control as possible. In compiling the control as it pertains to clause ordering, no significant changes occur in our proposed extension. Our objective in compiling higher-order unification is to generate instructions that mimic the WAM over first-order programs and extend naturally to higher-order programs. The key to realizing this objective is to recognize that SIMPL within backchaining steps can be compiled into WAM-like code. We illustrate this aspect below.

In compiling unification, an important concern is that of forming terms, particularly those involving abstractions and applications, on the heap. We introduce the following instructions for this purpose: store_apply, store_lambda, store_value, store_constant,
and \texttt{store\_de\_bruijn}. As an illustration of the use of these instructions, the term \((\lambda Y (g Y)) a\) will be compiled as follows.

\begin{verbatim}
store\_apply
store\_lambda
store\_apply
store\_constant g
store\_de\_bruijn 1
store\_constant a
\end{verbatim}

We also introduce the instruction \texttt{simpl} \(V_i,V_j\) which invokes \texttt{SIMPL} on \(V_i\) and \(V_j\) and is used for certain cases that cannot be dealt with entirely by compiled code. Further, there is an instruction called \texttt{finish\_unify}, which is generated at the end of a clause head and carries out a sequence of unification steps if the live FR list is nonempty. If the FR list is empty, \texttt{finish\_unify} acts as a no-op.

Our proposed \texttt{get} and \texttt{unify} instructions behave as in the WAM when their arguments are first-order terms. Their extension to higher-order terms is as follows:

\begin{enumerate}
\item If \(A_i\) points to a flexible nonvariable term, \texttt{get\_constant} \(C,A_i\) pushes the pair \(C,A_i\) on the FR stack, and \texttt{get\_structure} \(F,A_i\) sets the write mode and pushes on the FR stack a pair consisting of \(A_i\) and a pointer to the top of the heap. Finally \texttt{get\_value} invokes \texttt{SIMPL} and \texttt{get\_variable} remains unchanged.
\item In the write mode all \texttt{unify} instructions are unchanged; in the read mode, \texttt{unify\_constant}, \texttt{unify\_value} and \texttt{unify\_local\_value} invoke \texttt{SIMPL}, and \texttt{unify\_variable} remains unchanged.
\end{enumerate}

We note that, in the \texttt{get\_constant} \(C,A_i\) and \texttt{get\_structure} \(F,A_i\) instructions, the case when \(A_i\) points to a flexible nonvariable term is incorporated as the last case, just before backtracking. Hence the performance of these instructions on first-order arguments is slowed down by an extra test only if they initiate backtracking. We also note that \texttt{SIMPL} behaves identical to WAM's \texttt{get\_value} (i.e., would use the PDL) when its arguments are first-order terms, and would push terms on the FF and FR stacks only when its arguments are higher-order terms.

We illustrate the compiled code with two examples that focus on two different facets of the implementation. Consider first the program from Section 2 that is reproduced below.

\begin{verbatim}
mapfun [] F [].
\end{verbatim}
The generated WAM-like code for the example is shown below. Note that the first argument of the `switch_on_term` instruction corresponds to the case when register A1 points to a flexible term.

mapfun/3:  
  
  switch_on_term C1a, C1, C2, fail
  
  C1a:
  try_me_else C2a  % mapfun

  C1:
  get_nil A1  % []
  get_nil A3  % F, []
  finish_unify  %
  proceed  %

  C2a:
  trust_me_else fail  % mapfun

  C2:
  get_list A1  % []
  unify_variable X4  % X
  unify_variable A1  % [L1, F,]
  get_list A3  % []
  unify_variable X5  % S1
  unify_variable X3  % [L2]
  store_apply X6  % S1 = (%
  store_value A2  % F
  store_value X4  % X
  simplify X5, X6  % )
  finish_unify  % :-
  execute mapfun/3  % mapfun L1 F L2.

In generating the above code, we have assumed the following register allocation: variables L1, F, L2, X, S1 are allocated registers 1 ... 5 respectively. The mapfun example illustrates explicit use of higher-order features. An important feature of our implementation is that its performance matches the WAM implementation of Prolog when clauses are invoked with first-order terms. To substantiate this point, we show the compiled form of the `append` predicate, whose definition is reproduced below.

append [] Y Y.
append [H|T] Y [H|Z] :- append T Y Z.

Its compiled code is as follows.

append/3:  
  
  switch_on_term C1a, C1, C2, fail
  
  C1a:
  try_me_else C2a  % append

  C1:
  get_nil A1  % []
  get_value A2, A3  % Y, Y
  finish_unify  %
  proceed  %
Although the above code differs from normal WAM code for append only in the finish_unify, the reader should bear in mind that the get and unify have a different meaning. This enables the above definition for append to be applied to lists that have λ-terms embedded in them (assuming, as is the case in reality, that this definition is polymorphic). However, the behavior of this code over first-order lists practically mimics the execution of the WAM. Note that get_value and unify_value would use the PDL (pushdown list) in order to carry out the unification when their arguments are first-order terms, and no use would be made of the FF and FR stacks in these cases. The overhead of the finish_unify instruction is essentially one extra test per invocation of append to check if the FR list is empty. These observations lend support to our belief that the performance of our proposed abstract machine compares favorably with the WAM over first-order programs.

6. Conclusion

An execution model and an extended WAM instruction set for the compilation of higher-order Horn clauses were discussed in this paper. Several issues that do not arise in the implementation of first-order Horn clauses were addressed in this context: With respect to the execution model, the representation of λ-terms, branching within unification, and devices for the explicit representation of disagreement sets were discussed. With respect to compilation, we presented new instructions for creation of λ-terms, and, more importantly, showed how SIMPL within backchaining steps can be compiled using WAM-like instructions. A particular point to note is the manner in which the get and unify instructions were extended: the new instructions mimic the behavior of their counterparts in the WAM when their arguments

† This comparison of course ignores the cost of type checking necessary in our typed language.
are first-order terms while ensuring that the right actions are taken in
the presence of genuine higher-order terms. It is aspects such as this
that permit our abstract machine to be optimized to first-order pro-
grams. Furthermore we believe that techniques such as these may be
used to improve the behavior of an implementation on special cases of
higher-order unification without substantially degrading performance
in the general case.

Although the work described in this paper deals only with higher-
order aspects of \( hohh \) formulas, work is currently underway to extend
the proposed abstract machine to incorporate the new search opera-
tions. An effort to implement the abstract machine in this paper is
also being undertaken and will be extended to cover all the aspects of
\( hohh \) formulas. The actual implementation should, we believe, sup-
port an effort to fine-tune the abstract machine to situations that
occur often in practice. There is, thus, much work to be done before
an efficient and robust implementation of AProlog is provided. The
significance of this paper, however, is in that it takes a first step in
this direction and it describes machinery that will be germane to the
evolving effort.

7. Acknowledgements

We are grateful to Conal Elliott, Dale Miller and two anonymous
referees for comments on this paper. Support for this work from the
following sources is acknowledged: from NSF grant DCR-8603609 for
Jayaraman and NSF grant IRI-8805696 and ARO Contract DAAL03-
88-K-0082 for Nadathur.

8. References


of Symbolic Logic 5 (1940) 56 – 68.

[3] de Bruijn, N.G. Lambda calculus notation with nameless dum-
mies, a tool for automatic formula manipulation, with application
to the Church-Rosser Theorem. Proceedings, Series A, 75, No 5

[4] Huet, G. P. A unification algorithm for typed \( \lambda \)-calculus. Theo-
retical Computer Science 1 (1975) 27 – 57.

Proceedings of the Third International Logic Programming Con-


