CS-1993-24

# Implementing Polymorphic Typing in a Logic Programming Language

Keehang Kwon
Gopalan Nadathur
Debra Sue Wilson

Department of Computer Science

Duke University

Durham, North Carolina 27708–0129

September 1993

# Implementing Polymorphic Typing in a Logic Programming Language*

**Keehang Kwon, Gopalan Nadathur and Debra Sue Wilson**
Department of Computer Science
Duke University, Durham, NC 27706

**Abstract**

Introducing types into a logic programming language leads to the need for *typed* unification within the computation model. In the presence of polymorphism and higher-order features, this aspect forces analysis of types at run-time. We propose extensions to the Warren Abstract Machine (WAM) that permit such analysis to be done with reasonable efficiency. Much information about the structures of types is present at compile-time, and we show that this information can be used to considerably reduce the work during execution. We illustrate our ideas in the context of a typed version of Prolog. We describe a modified representation for terms, new instructions and additional data areas that in conjunction with existing WAM structures suffice to implement this language. The nature of compiled code is illustrated through examples, and the kind of run-time overheads that are incurred for processing types is analyzed, especially in those cases where others have shown that type checking can be eliminated during execution. The ideas presented here are being used in an implementation of the higher-order language called $\lambda$Prolog.

**Key Words:** Logic programming, typing, run-time type checking, implementation.

## 1 Introduction

There have been two different views of types in logic programming, manifest in the notions of *prescriptive* and *descriptive* typing [23]. The former notion corresponds to the use of a typed logic for programming, whereas the latter notion encapsulates an understanding of given (untyped) programs without changing the language used. At a pragmatic level, the first view of typing leads to languages with greater expressiveness while the second view corresponds to attaching information with programs that does not affect their meaning but that might be useful, for example, in improving their execution efficiency. Our interest in this paper is in the situation where a regimen

* This paper has been accepted for publication in *Computer Languages.* Comments on its contents are welcome and may be sent to the authors at the indicated addresses or, electronically, at the addresses `kwon@cs.duke.edu`, `gopalan@cs.duke.edu` or `dsw@cs.duke.edu`.

of prescriptive typing is adopted and, in particular, in the new implementation problems that arise in this context.

The main motivation for the work presented here is that of providing a good implementation for the higher-order logic programming language called $\lambda$Prolog [19] that, amongst other things, incorporates such a notion of typing [20]. Numerous applications have been discovered for this language over the last few years (*e.g.*, see [4, 5, 7, 14, 22]), stressing the importance of this concern. In providing a robust and efficient implementation for $\lambda$Prolog, three aspects that are novel to this language have to be dealt with: its higher-order features, its new search primitives and its typing regimen. The ideas discussed in this paper complement work pertaining to the first two aspects [10, 16, 17, 18, 21] and play a central role in an implementation that is being developed for this language. Despite the specific context of interest, we focus in this paper on a (typed) first-order sublanguage of $\lambda$Prolog. We make this choice so as to simplify the presentation, the higher-order features of the language being an aspect deserving treatment in their own right. The simplification nevertheless permits all the important aspects of an implementation to be discussed, and the ideas we present here are in fact used directly in an abstract machine that we have designed for the higher-order language [11]. We also note that the first-order language that is considered is itself of some interest, being closely related to typed versions of Prolog that have been proposed recently [8, 13, 15]. Finally, our work is also pertinent to languages with a different typing regimen — such as the one based on order-sorting [24] — and we believe that this relevance is more clearly exposed through the focus on a first-order language.

Typing is generally viewed as a device for providing information about the correctness of programs during compilation. One may therefore wonder why the use of types in logic programming would require any new mechanisms at run-time. However, the possibility that types might have to be examined during execution can be appreciated if one reflects on the fact that it is a *typed* logic that is now being used for programming. An immediate consequence of this change in language is that the computation process must use a form of *typed* unification. If a simplistic typing scheme is used, the new computation model can actually be implemented relatively easily by 'bundling' the type information into the names of symbols. Unfortunately, a useful type system must incorporate some form of polymorphism within it, and the specific type instances of symbols that are used in a computation can usually only be determined during execution. Further, precise knowledge of these type instances may be needed for several reasons. First, if the type system permits *ad hoc* polymorphism, the relevant definition for the procedure being invoked is dependent on this information. Second, in certain contexts, such as in $\lambda$Prolog, the unifiers for terms are determined by their types. Finally, the correct presentation of answers to queries requires not only the bindings for variables to be displayed, but also the right types.

The fact that the use of types within logic programming could lead to a need for examining them has been recognized previously. However, the emphasis has been on describing conditions under which type analysis can be avoided at run-time. (A notable exception is the work in [2] for an order-sorted language.) The general approach has been to identify situations in which success or failure in typed unification is not dependent on type annotations. One such situation, first presented in [15] and labeled in [8] as *type generality*, effectively amounts to banishing *ad hoc* polymorphism.[1] A generalization of this criterion is also described in [9]. While identifying such criteria is useful, this does not provide a scheme for dealing with situations in which the criteria described are not met. Further, the usual criteria presented for eliding types at run-time depend on a crucial property of a first-order language: types play a role in determining unifiability but do not influence the *structure* of unifiers. This property may not extend to other languages. In particular, it *does not* hold for a higher-order language such as $\lambda$Prolog. The specific computations that are performed in this language are intimately related to the types attached to the atomic symbols. Consequently, mechanisms are needed in such a language for determining and maintaining these types at run-time.

We consider in this paper additions to the usual machinery employed for implementing (untyped) Prolog for the purpose of providing an adequate treatment of types. The starting point for our implementation considerations is one that needs little justification: the basic paradigms of computation in the typed and untyped languages are very similar, the only difference being that *typed* unification must be accommodated. Our specific proposals are presented as modifications to the Warren Abstract Machine (WAM) [26]. For the immediate purpose of this paper, this choice is largely to provide a concreteness to the discussion. However, there is a definite reason for this preference in the context of a higher-order language. The choice is in a sense between a structure sharing and a structure copying approach, and efficiency in higher-order reduction dictates the latter. (The exact reasons are too detailed to describe here.) There is, of course, the question of whether any modification is needed to the basic scheme for the untyped language. As we shall see presently, a naive approach that can be adopted at least in the first-order context is to leave the usual machinery unchanged and to include types as an extra argument with function terms and procedure calls. However, this solution can be improved considerably by utilizing the type checking that is done during the compilation phase. In particular, corresponding to the type of every symbol a 'skeleton' can be identified that need never be checked or generated at run-time. Using this observation, a scheme can be devised that works correctly in all situations and exerts very little extra effort in the cases where others have shown that run-time type checking can be avoided.

The rest of this paper is organized as follows. In the next section we present a Prolog-like

[1] Actually, the exclusion of *ad hoc* polymorphism alone is not sufficient. A further restriction on function symbols is necessary — these symbols must be *type preserving* in the sense of [8].

language that incorporates a form of typing. We use this language to argue for the need for run-time type analysis in Section 3. In the following two sections, we describe enhancements to the WAM for implementing this language. Section 6 presents examples to illustrate the overall nature of our machine and offers some analysis of the scheme developed. Section 7 concludes the paper.

## 2 A Typed Version of Prolog

The language of interest to us is based on a variant of Horn clauses [25] that incorporates a type system similar in some ways to the one used in ML [6]. There are two components of this language that need to be described: the types and the terms and programs. The types of the language are constructed from a set $\mathcal{S}$ of *sorts*, a set $\mathcal{C}$ of *type constructors*, each member of which is specified with a fixed arity, and an infinite supply of *type variables*. The set $\mathcal{S}$ initially contains $o$, the type of propositions, and $int$, the type of integers, and $\mathcal{C}$ similarly contains the unary list type constructor $list$. The user may add to these collections by using declarations of a kind we do not further specify here. Type variables are distinguished by the usual Prolog convention for variables, i.e. they are denoted by tokens starting with an uppercase letter. The types are categorized into the *atomic types*, corresponding to sets of individual objects, and the types of functions and predicates. The atomic types consist of the sorts, the type variables and expressions of the form $(c\ \alpha_1\ \ldots\ \alpha_n)$ where $c$ is an $n$-ary type constructor and the $\alpha_i$s are atomic types. The types of functions are given by expressions of the form $\alpha_1 \rightarrow \ldots \rightarrow \alpha_n \rightarrow \beta$, where $\beta$ is an atomic type and each $\alpha_i$ is an atomic type other than $o$. Types $\alpha_1, \ldots, \alpha_n$ are referred to as the *argument types* of the function and $\beta$ is called its *target type*. In the case when the target type is $o$, the type in question is that of a predicate or procedure. We observe that a type has the structure of a first-order term. We make implicit use of this fact below.

The language of terms and programs is for the large part identical to that of Prolog with the exception that every expression now has an associated type. These types are obtained from associating, at the very lowest level, an atomic type with every constant and variable, a function type with every function symbol and a predicate type with each predicate symbol. The associations for constants and for function and predicate symbols are obtained, first of all, through a *defined* type for the symbol that may be provided by declarations such as in λProlog [20]. Particular occurrences of these symbols may then assume instances of the defined types as their types. The types of variables may be specified by an annotation such as $X\ :\ type$ at one of the occurrences of the variable in the expression; all the occurrences of the variable in the given expression must then adopt the specified type. Given an association of types with occurrences of atomic symbols, an expression such as $f(t_1, \ldots, t_n)$ is considered to be well-formed and to have the type $\beta_0$ just in case the type associated with (the occurrence of) $f$ is $\beta_1 \rightarrow \ldots \rightarrow \beta_n \rightarrow \beta_0$ and, for $1 \leq i \leq n$, $t_i$ is a well-formed expression of type $\beta_i$. To provide an illustration, let us assume that the defined type of 1 and 2 is $int$, of $cons$ is $A\rightarrow(list\ A)\rightarrow(list\ A)$, of $nil$ is $(list\ B)$ and of $append$ is $(list\ C)\rightarrow(list\ C)\rightarrow(list\ C)\rightarrow o$. Then the occurrence of $append$ and the two occurrences of $cons$ in the expression

$$append(cons(1, cons(2, nil)), (X : (list\ int)), (Y : (list\ int)))$$

can have the types $(list\ int)\rightarrow(list\ int)\rightarrow(list\ int)\rightarrow o$ and $int\rightarrow(list\ int)\rightarrow(list\ int)$, respectively. Further, if they do have these types, then the expression in question is well-formed and has the type $o$. However, if the type associated with the occurrence of the variable $Y$ in the expression is changed to $(list\ (list\ D))$, then the expression would not be well-formed.

Type variables result in a form of polymorphism in our language. A type that contains variables in reality stands for the infinite collection of types that are obtained by replacing the variables by ground types. This form of 'quantification' extends to a constant, variable, function symbol or predicate symbol that has the type associated with it. For instance, assume, as before, that $nil$ is a constant whose defined type is $(list\ A)$. Then it stands in reality for an infinite collection of constants each of which have the name $nil$ but whose associated types are $(list\ int)$, $(list\ (list\ int))$, and so on. Particular occurrences of constants, function symbols and predicate symbols may 'refine' the quantification implicit in their defined type. Thus, if an occurrence of $nil$ has the type $(list\ (list\ A))$ associated with it, then it stands for a collection that includes a constant with the type $(list\ (list\ int))$ but not one of type $(list\ int)$.

The implicit quantification provided for by type variables extends in a natural fashion to any term in whose type these variables might occur. Such an expression represents every well-formed expression that can be obtained by replacing each atomic symbol appearing in it by one that it represents; we note that the replacement for variables must be done in a consistent fashion. A particular kind of expression to which such a quantification applies is a program clause. Thus, consider the following clauses defining the $append$ predicate in which the occurrences of $append$, $cons$ and $nil$ have the types $(list\ A)\rightarrow(list\ A)\rightarrow(list\ A)\rightarrow o$, $A\rightarrow(list\ A)\rightarrow(list\ A)$ and $(list\ A)$ respectively:

$$\begin{array}{l} append(nil, (L : (list\ A)), L). \\ append(cons((X : A), (L1 : (list\ A))), (L2 : (list\ A)), cons(X, (L3 : (list\ A)))) \\ \qquad\qquad\qquad\qquad\qquad\qquad :\!-\ append(L1, L2, L3). \end{array}$$

Each of these clauses represents an infinite collection of clauses, the elements of these collections being obtained, effectively, by instantiating the type variable in the clause with a ground type. Viewed differently, the clauses actually define a polymorphic procedure that is capable of appending any two lists all of whose elements are of the same (ground) type. It is interesting to note that the polymorphism that is manifest in this example is *parametric*: the definition of appending lists remains the same regardless of the type of the elements of the list.

(1) $G_1, \ldots, G_{i-1}, p(t_1, \ldots, t_n), G_{i+1}, \ldots, G_m \Rightarrow$
$\varphi(G_1), \ldots, \varphi(G_{i-1}), \varphi(t_1) = \varphi(s_1), \ldots, \varphi(t_n) = \varphi(s_n),$
$\varphi(A_1), \ldots, \varphi(A_l), \varphi(G_{i+1}), \ldots, \varphi(G_m),$

if $p'(s_1, \ldots, s_n)$ `:-` $A_1, \ldots, A_l$ is a variant of a program clause in $\mathcal{P}$ whose free variables are chosen so as not to appear in the left-hand side of the rule, $p$ and $p'$ are identical except for their types and these types are unifiable and have $\varphi$ is a most general unifier.

(2) $G_1, \ldots, G_{i-1}, p(t_1, \ldots, t_n), G_{i+1}, \ldots, G_m \Rightarrow$
$\varphi(G_1), \ldots, \varphi(G_{i-1}), \varphi(t_1) = \varphi(s_1), \ldots, \varphi(t_n) = \varphi(s_n), \varphi(G_{i+1}), \ldots, \varphi(G_m),$

if $p'(s_1, \ldots, s_n)$ is a variant of a program clause in $\mathcal{P}$ whose free variables are chosen so as not to appear in the left-hand side of the rule, $p$ and $p'$ are identical except for their types and these types are unifiable and have $\varphi$ as a most general unifier.

(3) $G_1, \ldots, G_{i-1}, f(t_1, \ldots, t_n) = f'(s_1, \ldots, s_n), G_{i+1}, \ldots, G_m \Rightarrow$
$\varphi(G_1), \ldots, \varphi(G_{i-1}), \varphi(t_1) = \varphi(s_1), \ldots, \varphi(t_n) = \varphi(s_n), \varphi(G_{i+1}), \ldots, \varphi(G_m),$

if $f$ and $f'$ are identical except for their types and these types are unifiable and have $\varphi$ as a most general unifier.

(4) $G_1, \ldots, G_{i-1}, c = c, G_{i+1}, \ldots, G_m \Rightarrow G_1, \ldots, G_{i-1}, G_{i+1}, \ldots, G_m.$

(5) $G_1, \ldots, G_{i-1}, X = t, G_{i+1}, \ldots, G_m \Rightarrow \varphi(G_1), \ldots, \varphi(G_{i-1}), \varphi(G_{i+1}), \ldots, \varphi(G_m),$
provided $t$ is identical to $X$ or $X$ does not appear in $t$ and $\varphi$ represents the substitution of $t$ for $X$.

(6) $G_1, \ldots, G_{i-1}, t = X, G_{i+1}, \ldots, G_m \Rightarrow \varphi(G_1), \ldots, \varphi(G_{i-1}), \varphi(G_{i+1}), \ldots, \varphi(G_m),$
provided $t$ is identical to $X$ or $X$ does not appear in $t$ and $\varphi$ represents the substitution of $t$ for $X$.

Figure 1: State transition rules in the context of a given program $\mathcal{P}$

A query in the language being considered consists, as in Prolog, of a list of atoms. Answering such a query amounts, roughly, to finding a instance of these atoms that is satisfied by the given program. In defining the notion of computation precisely, some care must be exercised due to the presence of types. We provide a formal presentation of this notion by means of a transition system. The states of this transition system consist of lists of atoms or equations of the form $t_1 = t_2$, where $t_1$ and $t_2$ are terms in our typed language. Transitions between states are dependent on a particular program context and are given by the rules shown in Figure 1. The following tokens are used in these rules, perhaps with subscripts and superscripts, as schema variables for the indicated syntactic categories: $G$ for atoms or equations between terms, $A$ for atoms, $s$ and $t$ for terms, $p$ for predicate symbols, $f$ for function symbols, $c$ for constants and $X$ for variables. A rule of the form $l \Rightarrow r$ that appears in this figure is applicable to a state if it is an instance of $l$ and if the provisos on the rule are met. The application of such a rule results in a transition to the state that is the corresponding instance of $r$. In obtaining the new state, it may be necessary to make substitutions for some of the type or 'term' variables in a term, atom or equation. This is depicted in our transition rules by an expression of the form $\varphi(t)$ or $\varphi(G)$, where $\varphi$ represents the substitution. If $\varphi$ is a substitution for type variables, then its application consists of suitably modifying the types of the atomic symbols appearing in that expression.

We call a sequence of state transitions that start from a given query a *derivation*, and we depict it by listing the states that arise in the course of the transitions. Such a derivation is successful if it ends in a state given by the empty list. A successful derivation determines an instantiation for the variables in the original query and this is viewed, as usual, as an *answer* to the query. With respect to type variables in the query, this interpretation amounts to viewing them as being existentially quantified. An alternative description of this viewpoint that is in line with the earlier discussion of type variables is the following: A query in which type variables occur represents an infinite set of queries each member of which corresponds to instantiating these variables with ground types. An answer to any one of these instances is then also an answer to the overall query.

The objective is ultimately to find answers to queries. The notion of computation that is of interest may thus be described as a *search* for successful derivations. In structuring such a search, we need a method for determining which of several existing derivations to extend at a given point and for choosing between different transition rules that could be used to extend the selected derivation. The strategy that underlies the implementation described in this paper is the following. The search proceeds by always trying to extend a given derivation. This derivation may be characterized by the last state in it, which is referred to as the *current state*. In attempting to extend the derivation, only those rules are considered that pertain to the first element of the current state. If this is an atom, a program clause must be used to 'simplify' it. The rule that is used in this case is determined by the sequence in which these clauses appear in the program. A situation may, in general, be reached where all the rules that apply to the current state under the described restrictions have been exhausted. In this case, the search proceeds by retracting final segments of the existing derivation until a derivation is found whose last state can be transformed by some untried rule. If such a derivation is found, then the untried alternative is attempted. Otherwise the search ends in failure. The search procedure that results from adopting the strategy described is similar in structure to that underlying Prolog. The main difference is that it also takes into account the typed nature of our language.

We have assumed up to this point that the types of the various symbols that occur in a program are provided by the user. In reality this is not necessary. Given the defined types of constants, function symbols and predicate symbols and possibly the types of some occurrences of variables, a most general typing can be determined for all the symbols that occur in a given expression. We shall assume such a typing in the discussions that follow. In addition, the defined types of constants, function symbols and predicate symbols can themselves often be inferred in the form intended by the programmer by the process of reconstruction described in [20]. However, the details of type reconstruction are not very relevant to this paper. The notion of computation that is of interest requires a full knowledge of types at the points where the compilation and execution of programs are considered, and we shall assume that this is available in the remainder of this paper.

## 3 The Need for Run-Time Type Analysis

The model of computation for our language differs from that for Prolog mainly in that it uses a form of *typed* unification. To understand some of the implications of this difference, let us assume that $ft$ is a function symbol with defined type $A \rightarrow B \rightarrow int$ and consider the task of unifying the terms $ft(X, Y)$ and $ft(1, 2)$. To begin with, we observe that it makes sense to try to unify these terms only because they have the same types. Now, the existence of a unifier for these terms depends on the argument types of the occurrence of $ft$ in the first term. If these types are $int$, then the two terms have a unifier that is similar to the one obtained by ignoring the types. If the types are $i$ instead — we assume here and elsewhere in the paper that $i$ is a sort distinct from $int$ that is introduced by the programmer — then the terms are not unifiable. The types of symbol occurrences thus appear to affect the question of unifiability in an intrinsic way, indicating a possible need for the run-time processing of types.[2]

[2] The reader familiar with the language ML might wonder why a similar problem does not arise in that context. The reason for this is that in ML the variables $X$ and $Y$ in an expression such as $ft(X, Y)$ cannot assume a type more restrictive than the defined argument types of $ft$; in one sense, this is a consequence of the linearity restriction on patterns and the groundness requirement on expressions being evaluated. Thus, unification (or, more appropriately, matching) becomes independent of the specific types. A similar effect is obtained in the logic programming context by the type generality and type preserving conditions [8], but these restrictions have, in our opinion, less independent justification.

In a situation where only ground types are present, it is possible to bundle the type into the names of symbols, thereby eliminating the necessity to look at these explicitly during execution. Thus, in the case above, we may distinguish the function symbol $ft$ that has type $i \rightarrow i \rightarrow int$ from the function symbol with the same 'name' but type $int \rightarrow int \rightarrow int$, and this distinction may be used to determine the question of unifiability. However, the presence of polymorphism makes this kind of compile-time analysis insufficient in general. Since variables may occur in types, it is necessary to *unify* types, as opposed to merely checking for identity. The values that are thus determined for type variables may have a two-fold impact: they may be needed for understanding the answers that are found and also for determining which of several procedures are relevant for solving a subsequent goal.

To illustrate the latter possibility, we consider the following sequence of clauses assuming that the defined type of *print* is $A \rightarrow o$, the defined type of *print_list* and *print_list_aux* is $(list\ A) \rightarrow o$ and the defined types of *cons* and *nil* are as indicated in the previous section:

$$print(X : int) \text{ :- } write(X).$$
$$print(X : (list\ A)) \text{ :- } print\_list(X).$$
$$\ldots$$
$$print\_list\_aux(nil).$$
$$print\_list\_aux(cons(X, L)) \text{ :- } write(\text{','}), print(X), print\_list\_aux(L).$$
$$print\_list(nil) \text{ :- } write(\text{'[]'}).$$
$$print\_list(cons(X, L)) \text{ :- } write(\text{'['}), print(X), print\_list\_aux(L), write(\text{']'}).$$

The above definition of *print* exhibits a form of *ad hoc* polymorphism: the two clauses shown pertain to displaying integers and lists respectively and the elided part presumably contains clauses pertinent to displaying other kinds of objects. On the other hand, the definitions of *print_list* and *print_list_aux* are polymorphic in a parametric sense. The interesting aspect of these definitions is that they require type information to be available at run-time to ensure the correct processing of queries. Thus, the type of the object to be printed must be available during execution for the purpose of determining which of the several clauses for *print* is to be used. Notice that this also requires parametrically polymorphic definitions to process types. Thus consider the query

$$print\_list(cons(1, cons(2, cons(3, nil)))).$$

This query will in due course invoke *print_list_aux*. The latter procedure must determine that the type of the elements of the list that is being displayed is *int* and must pass this information on to *print* when it invokes that procedure.

The general observation in the examples above is that types are needed to determine the existence of unifiers and hence also the applicability of clauses. In the first-order context, types influence only the existence of unifiers and not their structure. It has therefore been suggested that analysis of types during execution may be eliminated in situations where the question of unifiability is itself independent of type information [9]. While this is of interest as an optimization technique, this is certainly not comprehensive enough to be an implementation strategy even for the first-order case. Moreover, the underlying assumption does not generalize to other contexts of interest, such as that of a higher-order language. In such contexts the *structures* of unifiers may also be dependent on typing. For example, let $F$ be a function variable and consider unifying the terms $a$ and $F(X)$ in a language such as $\lambda$Prolog. Assume $a$ has the type $i$. Now, if $F$ has the type $int \rightarrow i$, the most general unifier is given by the substitution for $F$ of the constant function which returns $a$, *i.e.* the lambda term $(\lambda y\ a)$. If the type of $F$ is $i \rightarrow i$ instead, there are two incomparable solutions. One of these binds $F$ to a constant function similar to that above (but with a different type), and the other binds $X$ to $a$ and $F$ to the identity function $(\lambda y\ y)$ on objects of type $i$. The choice of solutions is thus dependent on the type of $F$ and can be made correctly only if this type is present during computation.[3]

Typed unification can be rendered into untyped unification at least in the first-order context by including the type of function terms as an extra argument. Returning to the example at the beginning of the section and assuming that the types of $X$ and $Y$ are $i$, the attempt to unify the typed terms $ft(X,Y)$ and $ft(1,2)$ may be recast into an attempt to unify the untyped terms $ft(i \rightarrow i \rightarrow int, X, Y)$ and $ft(int \rightarrow int \rightarrow int, 1, 2)$. It may thus appear that any regular Prolog implementation would suffice for our typed language as well. There are, however, at least two drawbacks with this proposal. First, it does not generalize to the higher-order context for reasons mentioned earlier, and this is the context we are ultimately interested in. The second problem, and one that applies even to the first-order language under immediate consideration, is that this approach performs at run-time work that has already been performed during compilation. From the type declarations available at compile-time, it is known, for instance, that the type of *any* occurrence of $ft$ must have the structure $A \rightarrow B \rightarrow int$. During execution it is therefore only necessary to check the instantiations for $A$ and $B$ in determining whether particular occurrences of this function symbol are (or can be made) identical. The ideas that we present in the following sections utilize this observation to reduce the effort in processing types during the execution of the program.

## 4 A Machine Model Based on the WAM

An implementation of our language must include type information in the representation of terms. Further, space must be provided for creating new type expressions during execution and new mecha-

[3] A stronger statement can in fact be made: the kind of unification problem considered here can meaningfully be solved *only* in the presence of types. Thus, types in $\lambda$Prolog are not merely a device for indicating program correctness during compilation. They are in a sense necessary even for the existence of the language. We refer the reader to [20] for a fuller appreciation of this fact.

nisms are needed for performing the unification of types. We describe in this section the structures that must be added to the usual machinery employed for implementing logic programming languages for the purpose of accommodating these requirements. We base our discussions on one particular model, that of the WAM. The precise nature of the changes to this model depends on the choice of representation for types. In the first subsection below, we present a representation that follows up on the discussions in Section 3. We then describe the machinery that must be added to the WAM for the processing of types in light of this representation. We assume familiarity in these discussions with the basic issues pertaining to the WAM.

## 4.1 Representation of Typed Terms

Our representation of terms stores types only with occurrences of constants, variables and function symbols. This information uniquely determines the types of all terms and is sufficient for implementing the computation model described in Section 2. From the latter perspective, it is unnecessary even to store types with constants and variables: by the time unification 'descends' to constants or variables, types must have become identical. (It is as a result of this that types are not examined in rules (4)–(6) of Figure 1.) However, this information is maintained within the scheme we describe so as to permit the type of any arbitrary term to be determined in the course of execution. This ability is useful if, for instance, we wish to display the types of terms to the programmer. More importantly, this information is needed in the extention of our scheme to a higher-order language: types of variables and constants play a crucial role in a phase of higher-order unification that occurs after the analysis of the 'first-order' structure of terms contained in rules (3)–(6) in Figure 1.

In determining a suitable representation of types, it is useful to consider the manner in which these will be used. To understand this aspect, we return to the *append* predicate of Section 2 whose definition, with the typing of various symbols being left implicit, is reproduced below:

$append(nil, L, L).$
$append(cons(X, L1), L2, cons(X, L3))$ :- $append(L1, L2, L3).$

Consider now some of the tasks involved in evaluating a query such as

$append(cons(1, nil), cons(2, nil), R).$

Only the second clause is applicable in solving this query. In attempting to use it, it is necessary to unify the term $cons(1, nil)$ with $cons(X, L1)$. This task also involves checking if the type of *cons* in the first term is compatible with the expression $A \rightarrow (list\ A) \rightarrow (list\ A)$. This 'type checking' aspect must be made explicit in any compiled code that is generated for the second clause. In a similar fashion the term $cons(X, L3)$ must be 'constructed' and the variable $R$ bound to it. Constructing this term also involves processing types. For instance, it is necessary to construct the expression $A \to (list\ A) \to (list\ A)$ with a suitable binding for $A$ and to make this the type of the new instance of *cons* that is created. Once again the compiled code must account for this computation.

The above example shows that a considerable amount of work has to be done if the analysis of types is implemented in a naive fashion. Thus, the entire type of an occurrence of *cons* may have to be examined or constructed when it is encountered during execution. Luckily, most of this work becomes redundant with a suitable representation of types. As noted already, the static type checking phase ensures that every occurrence of *cons* actually has a type that matches the structure $A \to (list\ A) \to (list\ A)$. What distinguishes particular occurrences is the way the variable $A$ is (or must be) instantiated. Thus, if the type for any occurrence of *cons* is stored as the 'skeleton' $A \to (list\ A) \to (list\ A)$ plus the binding for $A$, the only work required during execution would be to check the binding for $A$ (during 'type checking') or to instantiate it (during 'type construction'). There is a further optimization that is applicable in the first-order context if types do not need to be displayed with answers: the type skeleton can be dispensed with. However, this optimization is not applicable in the higher-order case since the entire type of a symbol may have to be examined during unification. We therefore do not consider this optimization explicitly here.

The above discussion motivates representing a type by a skeleton and environment pair, where the skeleton is a pointer to a precomputed structure in a *type skeleton table* and the environment is a list of bindings for the variables appearing in the skeleton. The particular way a type is factored into these components in our implementation depends on the symbol whose type it is. For constants and function symbols, the skeletons that are used are their corresponding defined types. For a variable, the choice is dependent on the context in which the variable occurs: the argument type corresponding to the variable occurrence in the defined type of the first 'parent' (function or predicate) symbol is picked as the type skeleton for the variable. For instance, consider the term $p(X, f(X))$ assuming that the defined type of $p$ is $A \to A \to o$ and that of $f$ is $(list\ B) \to (list\ B)$. Our machine will use the $A$ as the type skeleton for $X$ in this case. An alternative is to use the most specific structure that is available for the type of the variable at compile-time. Thus, in the example considered, $(list\ B)$ could have been picked as the type skeleton for $X$. However, the first choice incurs fewer overheads within our overall processing scheme.

The reader may be tempted to construe the proposal just made as akin to advocating a structure sharing approach over the structure copying approach employed in usual WAM implementations. However, construing the proposal in this fashion is incorrect and also misses its central point. In the context of the first-order language, the main purpose of the separation of the type of a symbol into a skeleton and an environment is to distinguish between a part of the type that need *never* be examined during processing (but which is nevertheless needed to determine the full type of the symbol) and a part that may need to be examined. The latter component of the type, *i.e.*, the type environment may be represented in a variety of ways. The particular representation we use for it here is very similar to the representation of terms employed in, *e.g.*, [26], and in this sense shares with it the merits and demerits of the *structure copying* approach.

The actual implementation of our proposal differs from the 'schema' outlined only in that in the case of constants and function symbols the relevant type skeletons are directly associated with their names. To be precise, the representation of a term makes use of cells that are two words long and represent one of four different kinds of objects: function symbols, constants, (unbound) variables and references. The information stored in these cells is the following. There is, first of all, a tag distinguishing between the different kinds of data. For a function symbol, the additional information stored consists of the name, in the form of an address in the symbol table, and a pointer to the start of the type environment. The information retained for a constant is similar.[4] A variable contains a pointer to the type skeleton table and another pointer to the beginning of a type environment. Finally, a reference cell represents an indirection created by the binding of variables and stores additionally only the address of another heap location. The symbol table has a record for every function symbol and constant which includes, among other things, a pointer to its type skeleton in the type skeleton table. Type environments are written on a *type heap*, analogous to terms being written on the heap in the WAM. (The need for a separate type heap will become apparent in the next subsection and also when we consider compilation in Section 5.) The type environment of a term occupies a consecutive sequence of cells on the type heap with an entry for each distinct type variable occurring in the type skeleton associated with that term. We note that the specific representation used here differs from that in the WAM in that variables are distinguished from references. This division is somewhat more space efficient since combining references and variables necessitates cells that are three words long. However, operations such as trailing and dereferencing become slightly more expensive in time under this representation. While the trade-off could be made differently in the first-order context, certain other requirements determine the choice made here to be the more appropriate one in the higher-order case.

Figure 2 illustrates some of the details of our representation using the term $cons(cons(a, nil), L)$ and assuming that the defined type of $a$ is $i$ and of $cons$ is as indicated in Section 2.[5] In this figure we have made explicit also the representation of types in the type heap. We have chosen to use here the representation used in the WAM for terms as illustrated in, *e.g.*, [1]. Note that a compound term (structure) is represented as usual by placing the function symbol and arguments in consecutive cells on the heap. In conjunction with this figure, we observe that the depiction of the entries in

[4]Although we do not consider this aspect explicitly here, integers and floating point constants may be represented via specially tagged cells as in usual Prolog implementations. Type environment pointers will be unnecessary in both cases. In the case of integers this part of the cell may be unused or used to represent numbers in a larger range.

[5]The structure constructed to represent a term in a WAM-like context is dependent on the actual invocation pattern that leads to its construction. The representation shown here is only illustrative and not exclusive.

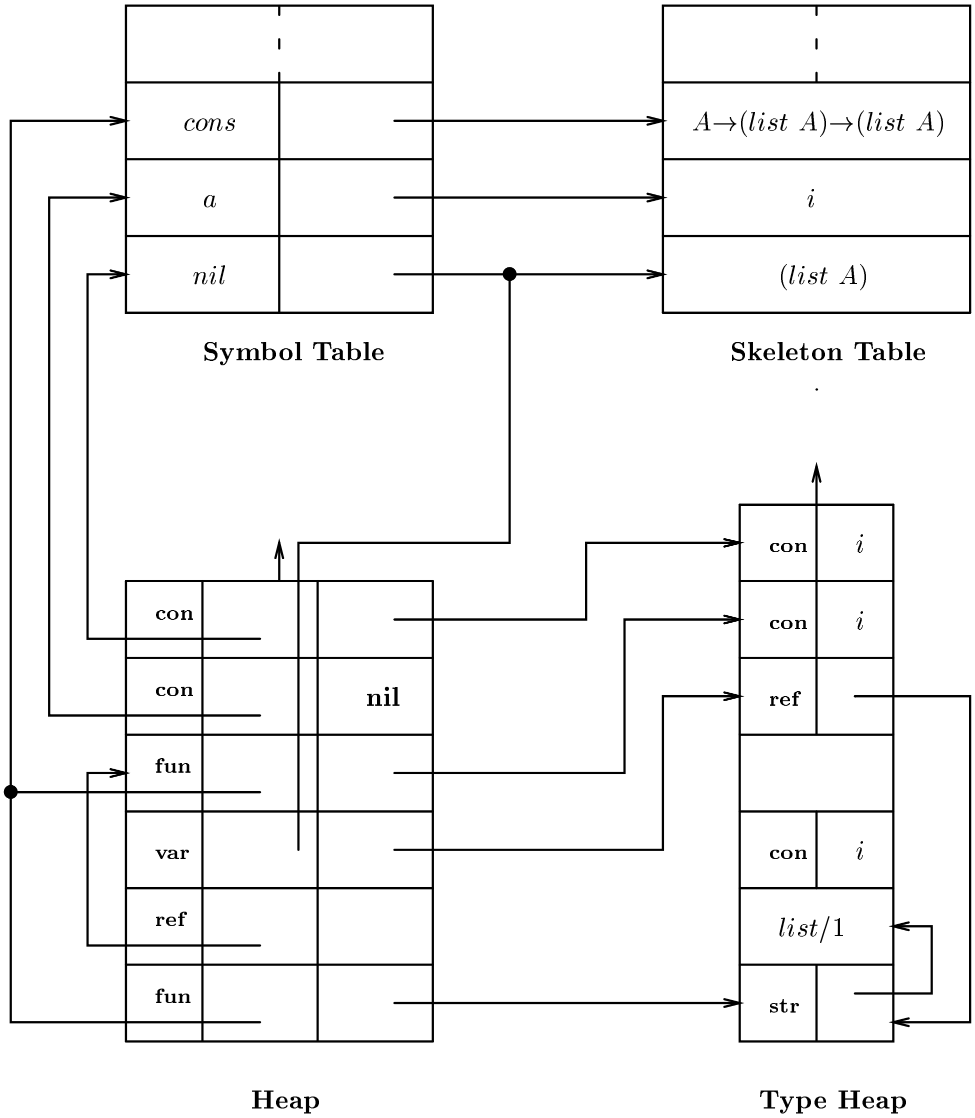


Figure 2: The representation of $cons(cons(a, nil), L)$ in the enhanced machine

the skeleton table is intended only to be schematic. In reality the types in each of the 'cells' will have to be represented explicitly. This is done in our machine by treating $\rightarrow$ as a binary function symbol and employing the usual representation for first-order terms. This representation parallels the one used for types in the type heap.

### 4.2 New Memory Components

Our machine model preserves most of the structures that appear in the WAM. However, some new devices are added to those present in the WAM for the purpose of processing types. We outline our enhanced machine below by describing only its new memory components, leaving the parts inherited from the WAM implicit. We also hint at the intended purpose of the new components. The precise manner in which these components are used will become clear when we consider the aspect of compilation.

The main addition to the structures of the WAM is that of a type heap. As the earlier discussions indicate, executing typed programs may require new type expressions to be constructed. In principle, these type expressions could be constructed on the WAM heap, and we have investigated this possibility. However, there are advantages to the divided heap space that we have adopted. In the usual implementation schemes for the untyped language, the manner in which compound terms are stored is a sensitive issue: the function symbol and the arguments must be stored in consecutive locations. The ordering of type checking and structure checking instructions is considerably constrained by this requirement if one heap is used. In particular, the interleaving of these kinds of instructions is precluded by this requirement. However, our processing scheme, as outlined at the beginning of Subsection 4.1 and as explained in greater detail in Sections 5 and 6, requires that these two kinds of instructions be interleaved. The division of space thus appears necessary for preserving the preferred representation of compound terms. With regard to the position of the type heap, we note that, in general, type expressions may be allocated either in the stack or in the type heap. The usual arguments relating to ease of backtracking then dictate that the type heap reside below the stack. Since our language of types is based on first-order terms, the structure of the type heap is similar to that of the heap in the WAM. (This observation also allows us to use a copy of the usual WAM instruction set in the type analysis process, as we explain in the next section.)

The other new components in our machine that play a role in the processing of types are summarized below:

| | |
|---|---|
| `type_S` | structure pointer to the type heap |
| `type_H` | register indicating the top of the type heap |
| `type_HB` | register indicating type heap backtrack point |
| `Z1,Z2, ...` | type registers |
| `type mode` | a mode with a read or write status |

These components, which parallel certain components that exist in the WAM, are used in implementing the unification of types. Their detailed use will become clear in the following sections. However, their purpose can be summarized as follows. The `Z` registers are aliases for the `A` or `X` registers that are used in the WAM for passing arguments or holding temporary values; this name is used only to distinguish the role of these registers in passing *type* arguments or in holding temporary *type* values. The `type_S` register plays a role in checking if the types in the type environment of an atomic symbol have a desired structure; in a certain sense, it is similar to the `S` register in the WAM. The purpose of the `type_H` register is self-explanatory, and the `type_HB` register serves a function that is analogous to that of the `HB` register in the WAM — it serves in determining what changes to type expressions might have to be undone and hence need to be trailed.

There are some additional points to be mentioned about our machine. Permanent type variables are allocated in it on the same stack as the permanent (term) variables, allowing WAM optimizations like last call optimization and environment trimming to be retained. Both the heap and the type heap expand with each procedure invocation, and contract on backtracking. A choice point record must now include a new field for recording the value of the `type_H` register. Finally, it is necessary to trail the bindings made for type variables in addition to those made for term variables. At a level of detail, it seems to be more (time) efficient to use separate trail stacks and separate methods for resetting type and term bindings, and this observation is reflected in the organization of our machine. However, a further discussion of this point is beyond the scope of this paper.

## 5 Compilation

One difference between compilation for our language and that for Prolog is that clauses must be examined for type correctness. A further difference is that the code that is produced must include instructions for performing type analysis at run-time. The structure of this code is based on the observation that typed unification in the first-order case can be rendered into untyped unification by including types as extra arguments in terms. (For the higher-order language, only a 'first-order like' part of the unification process is compiled and this observation applies to that part as well.) However, the compile-time checking and our representation of terms allow us to reduce the extra arguments to the *bindings* for type variables in the defined types of symbols. For example, let $p$ be a predicate symbol whose defined type is $A \rightarrow B \rightarrow o$. Then, an attempt to unify the two atoms

$p(X,Y)$ and $p(1,2)$ can be conceptualized as an attempt to unify the 'untyped' atoms $p(A,B,X,Y)$ and $p(int, int, 1, 2)$, assuming, of course, that the type of the occurrence of $p$ in the first atom is not a refinement of the defined type of $p$. This translation is used almost literally in treating procedure definitions and calls. Thus, new 'type' arguments are determined for predicates at compile time. The code for procedure calls must load the appropriate values into (type) argument registers and the code for procedure definitions must check the values of these arguments. However, a literal translation is not applied at the level of terms. For instance, consider an attempt to unify the typed terms $ft(X,Y)$ and $ft(1,2)$, assuming that the defined type of $ft$ is $A \rightarrow B \rightarrow o$. While this attempt can be visualized as an attempt to unify $ft(A,B,X,Y)$ and $ft(int, int, 1, 2)$, the terms continue to be represented as $ft(X,Y)$ and $ft(1,2)$ and the bindings determined for $A$ and $B$ affect only the type environment of the occurrence of $ft$ in the first term.

In implementing the above approach, changes and enhancements need to be made only to those WAM instructions that participate in unification. Consequently all other instructions are preserved unchanged in our machine. With regard to the instructions that are needed in unification, we note first that those that process function symbols must be modified so that they can initiate the processing of the new 'arguments'; these arguments are given by the type environment pointer associated with the incoming symbol or with the symbol to be created, and either reside on or must be put onto the type heap. Further, instructions which result in constants and variables being written on the heap must also initiate the writing of the associated type environments. Finally, new instructions are necessary for determining if the type arguments have the desired structure or, alternatively, for writing the bindings for these type arguments. We describe the modifications to the WAM instructions for unifying terms in greater detail below and then present the new instructions that are used for manipulating types. We use these instructions in the next section in illustrating the overall behavior of our enhanced machine.

### 5.1 Modifications to Instructions for Processing Terms

In addition to the preceding general remarks, the changes that are made to the original WAM instructions for the purpose of processing terms also incorporate a distinction between cases in which the symbol being processed is *monomorphic*, *i.e.*, its type skeleton contains no variables, and in which it is *polymorphic*, *i.e.*, its type skeleton contains one or more variables. It is necessary to initiate type checking if the symbol is polymorphic. To understand what exactly must be done in this case, consider an instance of the instruction `get_structure f,Ai` where `f` is polymorphic. If it is determined that register `Ai` references a structure whose head function symbol is `f`, then the type environment associated with this head must be located and the machine must prepare to do type unification. Alternatively, if `Ai` references an unbound variable, the machine must prepare to write the type environment to be associated with the structure `f`. However, none of this processing of types is necessary if `f` is monomorphic.

(1) As hinted earlier, the `get_structure` instruction splits into two forms, `get_p_structure` and `get_m_structure`, with the particular one to be used in a given situation depending on whether or not the relevant function symbol is polymorphic. A similar relation holds between `put_structure`, `put_p_structure` and `put_m_structure`. The desired action of the new instructions are explained as follows:

- `get_m_structure f,Ai` gets the value of register `Ai` and dereferences it. If the result is a reference to a structure with head function symbol `f`, the `S` register is set to point to the next address and execution proceeds in read mode. If the result is a reference to a variable, the variable is made into a reference to the top of the heap, and the binding is trailed if necessary. Further, the function symbol `f` with a `nil` type environment pointer is pushed onto the heap, and execution proceeds in write mode. In the remaining cases, backtracking is initiated.
- `get_p_structure f,Ai` gets the value of register `Ai` and dereferences it. If the result is a reference to a structure with head function symbol `f`, the `S` register is set to point to the next address and the `type_S` register is set to the type environment of `f`. Execution then proceeds in read and type read modes. If the result is a variable, the variable is made into a reference to the top of the heap, and the binding is trailed if necessary. Further, the function symbol `f` with a type environment pointer to the top of the type heap is pushed onto the heap and execution proceeds in write and type write modes. In the remaining cases, backtracking is initiated.
- `put_m_structure f,Ai` pushes the function symbol `f` with a `nil` type environment pointer onto the heap and puts a reference to it into `Ai`. Execution proceeds in write mode.
- `put_p_structure f,Ai` pushes the function symbol `f` with a type environment pointer to the top of the type heap onto the heap and puts a reference to it into `Ai`. Execution continues in write and type write modes.

(2) The instructions for compiling constants appearing in clauses obtain a treatment similar to that accorded to instructions dealing with structures. Thus, corresponding to each of the instructions `get_constant`, `unify_constant` and `put_constant`, there are now two instructions. One of these is used in compiling polymorphic constants and the other in compiling monomorphic constants. The mnemonic for these instructions is constructed in the same fashion as for the instructions for structures — e.g. the two instructions corresponding to

`get_constant` are `get_p_constant` and `get_m_constant`. The structures of all these instructions except `get_p_constant` and `unify_p_constant` are similar to those of their counterparts for the untyped language and the effects of these instructions can be understood by analogy to those for processing the head function symbol of structures. The instructions `get_p_constant` and `unify_p_constant` get an extra address argument and their interpretation is as follows:

- `get_p_constant c,Ai,L` gets the value of the register `Ai` and dereferences it. If the result is a reference to a variable, that variable is bound to the constant `c` with type environment pointer pointing to the top of the type heap, the binding is trailed if necessary and execution proceeds in type write mode. If the result is a constant with name `c`, execution jumps to address `L`. In the remaining case, backtracking is initiated.
- `unify_p_constant c,L`, when executed in read mode, dereferences the contents of the heap location pointed to by the `S` register. If the result is a reference to a variable, then that variable is bound to the constant `c` with type environment pointer pointing to the top of the type heap, the binding is trailed if necessary, the `S` register is incremented and execution proceeds in type write mode. If the result is a constant with name `c`, the `S` register is again incremented and execution jumps to address `L`; this allows the instructions which, in write mode, are required to copy the type environment of the variable to be skipped in read mode. In the remaining case, backtracking is initiated. In write mode, a constant with name `c` and type environment pointer set to the top of the type heap is written to the top of the heap and execution proceeds in type write mode.

(3) The instructions `put_variable` and `unify_variable` also give rise to two forms whose usage depends on whether or not the type skeleton associated with the variable has type variables in it. In the former case `put_p_variable` and `unify_p_variable` are used and in the latter case `put_m_variable` and `unify_m_variable` are used.

- `put_p_variable Xn,skel,Ai` creates a cell on the heap for an unbound variable and sets its type skeleton pointer to `skel`, which is itself a pointer to the type skeleton that is determined to be associated with the variable during compilation, and its type environment to point to the top of the type heap. A reference to the new cell is then put into registers `Ai` and `Xn` and execution proceeds in type write mode.
- `unify_p_variable Xn,skel,L`, in write mode, creates on the top of the heap an unbound variable whose type skeleton pointer is set to `skel` and whose type environment is a pointer to the top of the type heap. A reference to the newly created cell is then placed in `Xn` and execution proceeds in type write mode. In read mode, the contents of the heap location pointed to by the `S` register is stored in the `Xn` register, the `S` register is incremented and execution jumps to address `L`.

The instructions put_m_variable and unify_m_variable are identical to their polymorphic counterparts except that (i) the type environment field is set to *nil*, (ii) the manipulation of the type (read/write) mode is not necessary and (iii) in unify_m_variable, the label L is not used at all. Also, while we have discussed the modifications to put_variable and get_variable only in the context of temporary variables, the changes where permanent variables are involved are analogous.

(4) The following instructions are the same as in the WAM except that typed unification is used.

- get_value Vn,Ai
- unify_value Vn
- unify_local_value Vn

(5) 'Globalizing' a variable using unify_local_value Vn or put_unsafe_value Vn now requires copying the complete cell on the stack to the heap to prevent loss of type information.

We assume in this paper that there are no special instructions for handling lists; such instructions can, of course, be added by modifying the usual ones in a manner similar to that followed for instructions that process constants and function symbols. All other unification instructions that are not discussed explicitly here remain unchanged.

### 5.2 New Instructions for Processing Types

Unifying types essentially amounts to unifying first-order terms. Our extended machine, therefore, has a counterpart for each WAM instruction for the purpose of compiling type analysis. These new instructions are named by inserting '_type_' in the middle of the name of each WAM instruction. The only difference in their operation is that they work on the components related to types such as the type heap and the type_H and type_S registers. For example, put_type_variable Zn,Zi has the same effect as the WAM instruction put_variable Xn,Ai except that the required cell is created on the type heap instead of the heap.

Two further instructions are added to the instructions obtained from the WAM instructions in the fashion outlined above for the purpose of type unification: unified_type_value Vn and unified_type_local_value Vn. These instructions are used in place of unify_type_value and unify_type_local_value in situations when it is known during compilation that it is unnecessary to undertake unification when the instruction is executed in type read mode. Such a situation arises, for example, when a type variable in an argument in a structure also appears in the type of the functor of the structure. The instruction unified_type_value is a special version of the unify_type_value that in type read mode merely increments the type_S register by one. A similar relationship holds between the instructions `unified_type_local_value` and `unify_type_local_value`.

## 6 Some Examples and Analysis

We now illustrate the overall behavior of our machine and the use of our instructions by considering the compilation of some typed programs. As a first example, we consider compiling the clause

$$rel\_pair(X, Y) \text{ :- } rel(fst(X), fst(Y)), rel(snd(X), snd(Y)).$$

under the following assumptions: the defined types of $fst$ and $snd$ are $(pair\ A\ B)\to A$ and $(pair\ C\ D)\to D$ where $pair$ is a binary type constructor, the defined type of $rel$ is $E\to E\to o$ and the defined type of $rel\_pair$ is $(pair\ F\ G)\to(pair\ F\ G)\to o$. The predicate $rel\_pair$ has two type variables in its type skeleton. The assumption is that, on invoking this predicate, the bindings of these variables will be placed in the argument (`Z`) registers in much the same fashion as the 'real' arguments of the predicate are placed in the `A` registers. These values must be saved in choice point records for the same reason as must the usual arguments. Further, the values of type variables must be loaded into argument registers prior to calling a subgoal. These requirements dictate the structure of the compiled code for $rel\_pair$ that is shown below[6]:

```
allocate
get_type_variable Y3,Z1         % type variable A
get_type_variable Y4,Z2         % type variable B
get_variable Y1,A3              % X
get_variable Y2,A4              % Y
put_p_structure fst/1,A2        % (fst
unify_type_local_value Y3       % type variable A
unify_type_local_value Y4       % type variable B
unify_local_value Y1            % X)
put_p_structure fst/1,A3        % (fst
unify_type_local_value Y3       % type variable A
unify_type_local_value Y4       % type variable B
unify_local_value Y2            % Y)
```

[6]Given that type variables may also have to be stored in registers, it is natural to ask if there is some scheme for allocating registers. Although this question is not addressed explicitly in this paper, the structure of a general answer is not difficult to see. A compiler can easily make explicit the type arguments in the translation of clauses sketched at the beginning of Section 5. Standard register allocation algorithms, *e.g.*, the one presented in [3], can then be used. We assume such an algorithm in the compiled code presented here and in the other examples in this section.

```
call rel/2,4
put_type_unsafe_value Y4,Z1     % type variable B
put_p_structure snd/1,A2        % (snd
unify_type_local_value Y3       % type variable A
unify_type_local_value Y4       % type variable B
unify_local_value Y1            % X)
put_p_structure snd/1,A3        % (snd
unify_type_local_value Y3       % type variable A
unify_type_local_value Y4       % type variable B
unify_local_value Y2            % Y)
deallocate
execute rel/2
```

We would like to make some comparisons between the usual scheme that is employed for implementing an untyped logic programming language and the enhancements to such a scheme for dealing with a typed language that we have described here. For this purpose, we consider the compilation of the *append* predicate, the clauses for which are reproduced below:

$append(nil, L, L).$
$append(cons(X, L1), L2, cons(X, L3))$ `:-` $append(L1, L2, L3).$

Now, the comparison between the two schemes can be made at two different levels. At the first level, we could reduce our language, and consequently this definition, to an effectively untyped form and compare the code generated and the processing required under our scheme with the one presented, for example in [26]. At another level, we could preserve some of the advantages of typing and observe the overhead that this creates during execution.

An effectively untyped language can be obtained from our typed one under the following translation: we assume that there are only two sorts in the language, $i$ and $o$, that the type associated with each term is $i$ and that with propositions is $o$, and that the typing of constants, function symbols and predicate symbols are dictated by this requirement.[7] With regard to the *append* program above, the defined type of *append* under this reduction is $i \rightarrow i \rightarrow i \rightarrow o$ and *nil* and *cons* are assumed to have the defined types $i$ and $i \rightarrow i \rightarrow i$ respectively.

As a result of the typing scheme assumed above, variables do not occur in any types, *i.e.*, the constants, function symbols and predicate symbols are all monomorphic. Thus, all the needed type information is attached to these symbols at compile time under our implementation scheme, and

[7] The observant reader might object to this reduction on the grounds that our language was assumed at the outset to contain the sort *int* and the type constructor *list*. However, this objection is not serious: the reduction we describe here depends only on the user not employing *int* and *list* in constructing types in his/her programs.

virtually no type processing is done during execution. The compiled code that will be generated for *append* under this typing is displayed below, assuming that `list(1)` represents a pointer to the type skeleton $i$:

```
            switch_on_term C1a, C1, C2, fail
C1a:        try_me_else C2a
C1:         get_m_constant nil/0,A1          % nil
            get_value A2,A3                  % L, L
            proceed
C2a:        trust_me_else fail
C2:         get_m_structure cons/2,A1        % cons(
            unify_m_variable X4,list(1)      % X,
            unify_m_variable A1,list(1)      % L1)
            get_m_structure cons/2,A3        % cons(
            unify_value X4                   % X
            unify_m_variable A3,list(1)      % L3
            execute append/3
```

This code is very similar to that generated for an untyped language by the usual schemes. One difference is the use of the instructions `get_m_constant` and `get_m_structure` instead of `get_nil` and `get_list`. This difference is superficial and may, in fact, be eliminated. Another difference is that the occurrences of the `unify_m_variable` instructions must in write mode include a pointer to a type skeleton in the cell that is created. This additional operation does not have much time overhead. Yet another difference is that the instructions `get_value` and `unify_value` must perform typed unification. However, here too our scheme makes the overhead involved inconsequential: the only real overhead is incurred when polymorphic terms are processed, and a mere look at the type environment pointer suffices for determining whether or not a function, constant or variable symbol is polymorphic. Finally, our scheme involves a space overhead in that a type environment pointer must be included with each piece of data and a variable cell must also contain a pointer to a type skeleton. The former seems to be an unavoidable price to be paid for flexibility in typing. As for the latter, we have noted already that it is not necessary to include type information with variables in the context of a first-order language. (Thus the time overhead mentioned in connection with typed versions of `unify_variable` can also be eliminated.) However this is essential for a higher-order language and our implementation scheme includes this information because it is designed to eventually apply to such a language.

Rather than attempting to eliminate typing distinctions, we may actually make effective use of these. In such a situation, the *append* procedure may be interpreted as a polymorphic one, capable of appending lists of arbitrary, but homogeneous, type.[8] This capability is obtained by assuming that the defined types for the various symbols are the following: $(list\ A) \rightarrow (list\ A) \rightarrow (list\ A) \rightarrow o$ for *append*, $B \rightarrow (list\ B) \rightarrow (list\ B)$ for *cons* and $(list\ C)$ for *nil*. The compiled code that would be obtained for the definition of *append* under such a typing is given below. We assume here that `list(2)` and `list(3)` represent pointers to the type skeletons $B$ and $(list\ B)$, respectively.

```
       switch_on_term C1a, C1, C2, fail
 C1a:  try_me_else C2a
 C1:   get_p_constant nil/0,A2,T1          % nil
       unify_type_local_value Z1           % type variable A
 T1:   get_value A3,A4                     % L, L
       proceed
 C2a:  trust_me_else fail
 C2:   get_p_structure cons/2,A2           % cons(
       unified_type_local_value Z1         % type variable A,
       unify_p_variable X5,list(2),T2      % (X,
       unify_type_local_value Z1           % type variable A),
 T2:   unify_p_variable A2,list(3),T3      % (L1,
       unify_type_local_value Z1           % type variable A)), L2,
 T3:   get_p_structure cons/2,A4           % cons(
       unified_type_local_value Z1         % type variable A,
       unify_value X5                      % X,
       unify_p_variable A4,list(3),T4      % (L3,
       unify_type_local_value Z1           % type variable A))
 T4:   execute append/3
```

In contrast to the monomorphic case, this code includes several instructions that participate in typed unification. However, it is interesting to note that very little overhead is incurred by these instructions when the unification of terms takes place in read mode. All the `unify_type_local_value` instructions are skipped over in this case. There are two more instructions that are pertinent, namely the `unified_type_local_value` instructions. While these are not skipped over, their effect in read mode is merely to increment a register. There are some overheads when unification of terms proceeds in write mode because the machine must build the type environments for the various structures that it is creating. However, even these operations are not a major penalty on

[8] The advantages of using typing in this case might be obvious. A general discussion of this issue also occurs in [20].

performance since no type expressions are actually built. In fact only pointers to prior type expressions are pushed onto the type heap. Along another dimension, considerable (type) structure can be shared between the type environment of a function symbol and those of its arguments, and this helps reduce the space overhead in general.

The typing assumed for the various symbols in the last example satisfies the type generality and type preserving properties of [8]. The analysis in [8] (and in [15]) shows that success and failure in a typed first-order language satisfying these properties is independent of specific typing information. Thus, a typed first-order language in which these conditions are guaranteed to be satisfied can, in a sense, be implemented as an untyped language. The scheme that we have proposed clearly incurs an overhead over such an implementation. The nature of this overhead has been exposed and discussed in the case of the polymorphic version of *append*, and a similar analysis can be provided in other situations where the type generality and type preserving conditions are satisfied. While the significance of the overhead can be debated, we note that our scheme has the advantage of being completely general: it yields an implementation even in situations where the type generality (or related) condition is not satisfied. Such situations could arise naturally in a first-order language, as exemplified in Section 3. More importantly, our scheme carries over readily to the implementation of higher-order languages. Typing information must be explicitly present in these languages for determining the set of solutions and it is difficult to see how the overhead incurred by our implementation method can be avoided in such a context.

# 7 Conclusion

We have discussed in this paper the implementation of a logic programming language that incorporates a form of polymorphic typing. Our main objective was to examine ways in which the analysis of types at execution time could be minimized. We have suggested the following ways in which this goal can be achieved:

(1) By using a representation of types that separates the information already present during compilation from that which must be determined during execution. This permits less work to be done in checking types at run-time.

(2) By introducing instructions for compiling as much of the remaining type checking as is possible, thereby eliminating the need to do this via code run in interpretive mode.

We have illustrated the applicability of these ideas by considering the implementation of a typed version of Prolog that is closely related to others that have been proposed recently (*e.g.*, see [8]). The ideas presented here can also be used in the context of other typed languages. They are applicable, for instance, to a higher-order language and we have also used them in [12] to describe an alternative implementation scheme for the polymorphic order-sorted language considered in [2].

We recall that the original motivation for this work was that of implementing the higher-order language called λProlog. The ideas presented here are in fact employed more or less directly in an abstract machine that we have devised for this language that also incorporates devices for handling higher-order features [16, 18, 21] and for implementing scoping constructs in logic programming [10, 17]. An emulator for this machine is currently being implemented and we believe that this effort will provide a practical vindication for the ideas described in this paper.

## 8 Acknowledgements

We are grateful to Bharat Jayaraman for suggestions provided at an early stage in this work and for his comments on an earlier version of this paper. Some suggestions for improvement were also provided by anonymous referees. Support for this research was obtained from NSF grant CCR-89-05825 and CCR-92-08465.

## References

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

[2] C. Beierle, G. Meyer, and H. Semle. Extending the Warren abstract machine to polymorphic order-sorted resolution. In Vijay Saraswat and Kazunori Ueda, editors, *Proceedings of the International Logic Programming Symposium*, pages 272–286. MIT Press, 1991.

[3] Saumya K. Debray. Register allocation in a Prolog machine. In *Proceedings of the 1986 Symposium on Logic Programming*, pages 267–275. IEEE Computer Society Press, September 1986.

[4] Scott Dietzen and Frank Pfenning. Higher-order and modal logic as a framework for explanation-based generalization. *Machine Learning*, 9:23–55, 1992.

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

[6] Michael J. Gordon, Arthur J. Milner, and Christopher P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.

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

[8] Michael Hanus. Horn clause programs with polymorphic types: Semantics and resolution. In J. Diaz and F. Orejas, editors, *TAPSOFT 89*. Springer-Verlag, 1989. Lecture Notes in Computer Science Vol 352.

[9] Michael Hanus. Polymorphic higher-order programming in Prolog. In Giorgio Levi and Maurizio Martelli, editors, *Proceedings of the Sixth International Logic Programming Conference*, pages 382–398. MIT Press, 1989.

[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 and Gopalan Nadathur. An instruction set for higher-order hereditary Harrop formulas. In *Proceedings of the Workshop on the λProlog Programming Language*, Philadelphia, 1992. (to appear).

[12] Keehang Kwon, Gopalan Nadathur, and Debra Sue Wilson. Implementing logic programming languages with polymorphic typing. Technical Report CS-1991-39, Computer Science Department, Duke University, 1991.

[13] T.K. Lakshman and U.S. Reddy. Typed Prolog: A semantic reconstruction of the Mycroft-O'Keefe type system. In Vijay Saraswat and Kazunori Ueda, editors, *Proceedings of the International Logic Programming Symposium*, pages 202–217. MIT Press, 1991.

[14] 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.

[15] A. Mycroft and R. A. O'Keefe. A polymorphic type system for Prolog. *Artificial Intelligence*, 23:295–307, 1984.

[16] Gopalan Nadathur and Bharat Jayaraman. Towards a WAM model for λProlog. In Ewing Lusk and Ross Overbeek, editors, *Proceedings of the North American Conference on Logic Programming*, pages 1180–1198, Cleveland, Ohio, October 1989.

[17] Gopalan Nadathur, Bharat Jayaraman, and Keehang Kwon. Scoping constructs in logic programming: Implementation problems and their solution. Technical Report CS-1993-17, Department of Computer Science, Duke University, July 1993.

[18] 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.

[19] 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.

[20] Gopalan Nadathur and Frank Pfenning. The type system of a higher-order logic programming language. In Frank Pfenning, editor, *Types in Logic Programming*, pages 245–283. MIT Press, 1992.

[21] Gopalan Nadathur and Debra Sue Wilson. A representation of lambda terms suitable for operations on their intensions. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 341–348. ACM Press, 1990.

[22] Frank Pfenning. Partial polymorphic type inference and higher-order unification. In *Proceedings of the ACM Lisp and Functional Programming Conference*, pages 153–163, 1988.

[23] Uday S. Reddy. Notions of polymorphism for predicate logic programs. In K. Bowen and R. Kowalski, editors, *Proceedings of the Fifth International Conference and Symposium on Logic Programming.* MIT Press, 1988.

[24] G. Smolka. TEL (Version 0.9), report and users manual. Technical Report SEKI-Report SR 87-17, FB informatik, Univ. Kaiserslautern, 1988.

[25] M. H. van Emden and R. H. Kowalski. The semantics of predicate logic as a programming language. *Journal of the ACM*, 23(4):733–742, 1976.

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