Optimizing the Runtime Processing of Types in a Higher-Order Logic Programming Language

Gopalan Nadathur and Xiaochu Qi

Department of Computer Science and Engineering, University of Minnesota,
4-192 EE/CS Building, 200 Union Street SE, Minneapolis, MN 55455
`Email: {gopalan,xqi}@cs.umn.edu, Fax: 612-625-0572`

Abstract. The traditional purpose of types in programming languages of providing correctness assurances at compile time is increasingly being supplemented by a direct role for them in the computational process. In the specific context of typed logic programming, this is manifest in their effect on the unification operation. Their influence takes two different forms. First, in a situation where polymorphism is permitted, type information is needed to determine if different occurrences of the same name in fact denote an identical constant. Second, type information may determine the specific form of a binding for a variable. When types are needed for the second purpose as in the case of higher-order unification, these have to be available with every variable and constant. However, in many situations such as first-order and higher-order pattern unification it turns out that types have no impact on the variable binding process. As a consequence, type examination is needed in these situations only for the first of the two purposes described and even here a careful preprocessing can considerably reduce their runtime footprint. We develop a scheme for treating types in these contexts that exploits this observation. Under this scheme, type information is elided in most cases and is embedded into term structure when this is not entirely possible. Our approach obviates types when properties known as definitional genericity and type preservation are satisfied and has the advantage of working even when these conditions are violated.

1 Introduction

This paper concerns the runtime treatment of types in a higher-order logic programming language that incorporates polymorphic typing. We are interested in a setting where types are used prescriptively, i.e., where their purpose is to impose coherence conditions on expressions in a program. The traditional utility for such conditions is to express limitations in the applicability of specific operations, thereby providing a control over the kinds of computations that are attempted. This is, in fact, a role for types that is relevant to program correctness and one that is typically discharged at compile-time. There is, however, another mode in which types can be used: they can be employed to influence the kind of computation that is carried out. Such a usage of types leads to *ad hoc* polymorphism, a facet that is exploited systematically in object-oriented programming and also sometimes imported into functional programming contexts for efficiency reasons [15]. It is when types are used in this fashion that they exhibit a runtime presence.

The two uses of types that we describe above apply also in the logic programming setting; they are present, for instance, in the language λProlog [12]. The runtime effects of types are characterized within this paradigm by their role in the unification operation. This operation is carried out by a possibly repeated application of two phases. One of these phases is that of *term simplification*, a critical part of this computation being that of matching the constants at the heads of the two terms that are being unified. In a polymorphic setting, different instances of a constant with a particular name may have distinct associated types and information must be available for determining if these can be made identical. The other phase is one in which a binding is determined for a variable that appears at the head of one of the terms. Types can affect this *variable binding* phase as well, impacting thereby on the shape of unifiers rather than merely on the question of unifiability. When types influence both phases, as they do in the case of higher-order unification [3], they must be available with each variable and constant appearing in a term.

Types typically have a rich structure in declarative programming languages, making their runtime processing a costly operation. The usual resolution to this problem in the typed logic programming setting is to restrict the language so as to altogether eliminate their need in computations. The language that is at the center of most such proposals is either a first-order one or, at least, uses unification in a first-order way. In such a situation, types can be made irrelevant to the variable binding phase. Conditions are then imposed on the structure of the declared types of constants, the instance types of the predicates that appear as the head of clauses and possibly on the mode in which predicates are used to ensure that types are not needed to determine unifiability either. Exemplars of this approach are those presented in [1, 2, 5, 9].[1]

Our concern in this paper also is to minimize the impact of types on runtime behaviour. However, we take the view that we cannot change the language to suit our needs as its implementors. Instead, we focus on a combination of compile-time analysis and a processing structure that can reduce the runtime footprint of types. The key ingredients of our approach are the following:

- We orient our implementation around a form of unification in which types do not impact on the variable binding phase; this allows us to elide types with variables.
- Following [4], we utilize information available from signature declarations to factor types for constants into a fixed skeleton part that we discard and a variable part that we carry around at runtime.

[1] Both [1] and [2] seem to suggest that their conditions can be applied on a "per constant" and "per clause" basis. However, the proposals in these papers are incorrect if interpreted in this way; see Section 5 for a specific example to this effect.

– Using a compile-time examination of predicate definitions and the structure of the types for constants, we isolate and eliminate those variable parts in types over which unification is guaranteed to succeed.

The scheme we describe allows all runtime computations over types to be eliminated when the conditions known as *type generality* and *type preservation* required by many of the previously described approaches are met and degrades gracefully to function also in situations where these are not satisfied.

The rest of this paper is organized as follows. In the next two sections we describe the typed language and we present a computational model for it around which we orient our implementation ideas. In Section 4 we show how compile-time type checking and the structure of types can be exploited to eliminate much of the type information with non-predicate constants. These methods are not quite as useful for predicate constants. For such constants, we have to analyze the usage of type information in goal invocations, an aspect that we discuss in Section 5. We conclude the paper in Section 6 with an indication of how the ideas that are presented in it are actually being used.

2 The Syntax of the Typed Language

We consider the core language of λProlog in this paper with a restriction: we do not permit predicate quantification and we disallow predicates and logical symbols within the arguments of predicates. This omission simplifies our presentation without seriously limiting the applicability of the scheme that we develop.

The types that are used are similar to the ones employed in a language such as SML. We begin with sorts and type variables and use type constructors to build structured types over these. We assume a collection of built-in sorts such as *int*, *string*, and o (that stands for propositions) and the well-known unary type constructor *list*. Syntactically, type variables are distinguished as tokens that begin with uppercase letters. Using this vocabulary, we obtain types such as *int*, *(list int)* and *(list A)*. The last is an example of a polymorphic type whose different manifestations are obtained by suitably instantiating the variable A. Existing collections of sorts and type constructors can be enhanced through mechanisms whose details we omit. We use a curried syntax for constructed types. Thus, if *pair* has been identified as a binary type constructor, then the expression *(pair int string)* is a type; note that a constructor must be given a number of arguments equal to its arity to produce a legitimate type. The types that we have described thus far constitute *atomic* types. The language also admits of *function* types, written as $\alpha \rightarrow \beta$ where α and β are types. Parentheses are omitted in type expressions by assuming that $\rightarrow$ associates to the right. Using this convention, a function type may be depicted in the form $\alpha_1 \rightarrow \cdots \rightarrow \alpha_n \rightarrow \beta$ where β is an atomic type. Such a type has $\alpha_1, \ldots, \alpha_n$ as its *argument* types and β as its *target* type. This notation and terminology is extended to atomic types by allowing the argument types to be missing. We do not permit o to appear in argument types.

The terms of the language are those of a lambda calculus restricted by the types just described. The starting point is provided by a collection of constants and variables each element of which has a designated type. We assume as built-in the usual integer and string constants of type *int* and *string* and the list constructors *nil* of type *(list A)* and *::* of type *(A* $\rightarrow$ *(list A)* $\rightarrow$ *(list A))*, the latter being written as an infix, right associative operator. Additional constants can be identified together with their types in a manner that we do not detail here. In constructing terms, we are permitted to use constants at instances of their defined types. In particular, the terms are given with their associated types by the following rules: (i) a variable is a term of its associated type, (ii) a constant is a term of any instance of its defined type, (iii) if t and s are terms of type $\alpha \rightarrow \beta$ and α respectively, then $(t\ s)$ is an (application) term of type β, and (iv) if x is a variable of type α and t is a term of type β, then $\lambda x\, t$ is an (abstraction) term of type $\alpha \rightarrow \beta$. In writing terms, we shall use the conventions that application associates to the left and has higher priority than abstraction.

We assume a notion of equality on terms that is given by the rules of α-, β- and η-conversion. Types ensure that these rules can be used to convert every term into a *head-normal form.* Such a form has the structure $\lambda x_1 \ldots \lambda x_n\, (h\ t_1\ \ldots\ t_n)$, where h is a constant or variable; we shall refer to h as the head of the term and to $t_1, \ldots, t_n$ as its arguments. We also observe that, given two head-normal forms of the same type, the α- and η-rules allow us to arrange the abstractions at the front to be identical in number and in the names for the bound variables. We utilize these facts implicitly in the discussions that follow.

Programming in the language is based on two sets of formulas called *program clauses* and *queries* or *goals.* Formulas in these two classes are constructed using logical symbols from atomic ones that are actually terms of type o with (predicate) constants as head. Denoting atomic formulas by the symbol A and using x to represent variables that do not have o as a target type, program clauses and goals are the D and G formulas given by the following syntax rules:

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

Computation consists of attempting to solve a closed query relative to a collection of closed program clauses in a manner that we explain in the next section.

We will use devices familiar from Prolog when we have to depict actual programs. In particular, we will adopt Prolog's manner for writing implications in program clauses, its convention of making top-level universal quantifiers implicit by using names beginning with uppercase letters for quantified variables and its method for depicting sets of clauses. As an illustration, the program

$$\begin{aligned} \{\ & \forall l\ \textit{(append nil l l)}, \\ & \forall x\, \forall l1\, \forall l2\, \forall l3\ \textit{((append l1 l2 l3)} \supset \textit{(append (x::l1) l2 (x::l3)))}\ \}, \end{aligned}$$

in which we assume *append* to be a predicate constant of type

(list A) $\rightarrow$ *(list A)* $\rightarrow$ *(list A)* $\rightarrow$ o,

will be rendered as

> *append nil L L.*
> *(append (X::L1) L2 (X::L3)) :- (append L1 L2 L3).*

Similarly, the convention for making top-level existential quantifiers in queries implicit will also be used. Thus, the query

> $\exists f\, \forall a$ *(append (a::nil) (b::a::nil) (f a))*,

where we assume b to be a constant of a (new) type i, will be depicted as

> $\forall a$ *(append (a::nil) (b::a::nil) (F a)).*

Solving a query is intended to produce a bindings for its implicitly quantified variables. Thus, in this instance, the result would be the binding λx *(x::b::x::nil)* for F. This query incidentally illustrates the fact that the language is higher-order and that computation in it can take place under a mixed prefix of quantifiers.

We have thus far been silent about the manner in which types are associated with variables. A means for doing this may be included with the abstractions and quantifiers that introduce them in terms. It is also possible to infer a unique most general type for them using ideas familiar from SML; using this approach we would, for instance, infer the type *(list A)* for the variable L that appears in the first clause for *append* above. For constants, we have to contend with the fact that their defined types may be refined in specific contexts of use; this happens for instance for both *::* and *nil* in the term *(1::nil)*. In the end, these specific type associations may have to be carried into computations. We shall depict them as subscripts on variables and constants in the next section when we spell out the evaluation model. We then devote our attention to the efficient treatment of these type annotations.

3 The Model of Computation

Given a program $\mathcal{P}$, let us denote the set of instances of clauses in $\mathcal{P}$ obtained by substituting ground types for the type variables appearing in them by $\{\mathcal{P}\}_t$. Similarly, let us denote the set of all ground type instances of a goal G by $\{G\}_t$. A goal G is then intended to be solvable from a program $\mathcal{P}$ if and only if there is a $G' \in \{G\}_t$ such that $\{\mathcal{P}\}_t \vdash G'$ holds in intuitionistic logic. Our language possesses the uniform provability property [8] and this fact allows us to use a procedure similar to the one for Prolog in addressing this derivability question. In particular, given a complex goal, we may proceed by simplifying it as per its top-level logical symbol. When this symbol is an existential quantifier, we introduce a special *logic* variable that serves as a place-holder for a term whose precise shape will be determined as the search proceeds. When the goal has been reduced to an atomic one, we use clauses from the program in a backchaining mode. This step makes use of unification and may yield a further goal to solve, leading to a repetition of the overall process.

There are, however, new aspects to be dealt with arising out of the richer syntax of our language. One such aspect relates to the possible presence of implications in goals. The program can change dynamically because of this and the solution of each subgoal must therefore be relativized to its own program. Another issue concerns the treatment of mixed prefixes of quantifiers. Universal quantifiers in goals lead to the introduction of new constants during computation and unification must be designed to respect the scope of such constants. To realize this requirement in an interpreter, we think of annotating each constant and logic variable with a level indicator and using these annotations in an occurs-check phase in unification.

$$\frac{\theta \in unify(A, A')}{\mathcal{P}, n \vdash A, \theta}\ [\text{ATOM}] \qquad \frac{\theta \in unify(A, A') \qquad \theta(\mathcal{P}), n \vdash \theta(G), \theta'}{\mathcal{P}, n \vdash A, \theta'(\theta)}\ [\text{BC}]$$

where $A' \in [\mathcal{P}]_n$ where $G \supset A' \in [\mathcal{P}]_n$

$$\frac{\mathcal{P} \cup \{D\}, n \vdash G, \theta}{\mathcal{P}, n \vdash D \supset G, \theta}\ [\text{IMP}] \qquad \frac{\mathcal{P}, n \vdash G_1, \theta' \qquad \theta'(\mathcal{P}), n \vdash \theta'(G_2), \theta}{\mathcal{P}, n \vdash G_1 \wedge G_2, \theta}\ [\text{AND}]$$

$$\frac{\mathcal{P}, n \vdash G[x := X^n], \theta}{\mathcal{P}, n \vdash \exists x\, G, \theta}\ [\text{SOME}] \qquad \frac{\mathcal{P}, n+1 \vdash G[x := c^{n+1}], \theta}{\mathcal{P}, n \vdash \forall x\, G, \theta}\ [\text{ALL}]$$

where X is a new logic variable of the same type as x where c is a new constant of the same type as x

Fig. 1. The operational semantics rules

Towards implementing these ideas, we allow logic variables to appear in our formulas and we label each such variable and constant with a natural number; we display this label where needed below as a superscript on the corresponding symbol. We then orient the operational semantics of our language around the derivation of judgements of the form $\mathcal{P}, n \vdash G, \theta$, where $\mathcal{P}$ is a program, n is a natural number, G is a goal and θ is a substitution for logic variables and type variables. Let us write $F \in [\mathcal{P}]_n$ if F can be obtained from a clause in $\mathcal{P}$ by first picking fresh names for the type variables that appear in it and then instantiating the universal quantifiers that appear at its head with new logic variables carrying the label n. Moreover, let us denote the result of replacing the variable x in a formula F with t by the expression $F[x := t]$. Then the rules shown in Figure 1 allow us to derive the judgements that are of interest to us. To solve the (top-level) goal G from the program $\mathcal{P}$, we label all the constants appearing in G and in $\mathcal{P}$ with 0 and then try to construct a derivation for $\mathcal{P}, 0 \vdash G, \theta$ for some θ using these rules. Notice that the substitution component of such a judgement actually constitutes the result produced by a computation and, when thought of in this manner, this imposes a sequentiality in the solution of conjunctive goals using the rule [AND].

The rules in Figure 1 rely on a unification judgement. In elaborating this, we shall assume that all the unification problems that we encounter dynamically satisfy the following restriction: whenever a logic variable appears as the head of

(1) $\langle(\lambda x\, t = \lambda x\, s :: E, \theta\rangle \longrightarrow \langle(t = s) :: E, \theta\rangle$.

(2) $\langle(a_\tau\ t_1\ \ldots\ t_n) = (a_\sigma\ s_1\ \ldots\ s_n) :: E, \theta\rangle$
$\longrightarrow \langle\phi(t_1 = s_1 :: \ldots :: t_n = s_n :: E), \phi \circ \theta\rangle$,
provided a is a constant or a variable bound by an abstraction and ϕ is a most general unifier for τ and σ

(3) $\langle((F_\sigma\ y_1\ \ldots\ y_n) = t) :: E, \theta\rangle \longrightarrow \langle\varphi(E), \varphi \circ \theta\rangle$
provided F is a logic variable and $mksubst(F_\sigma, t, [y_1, \ldots, y_n]) = \varphi$.

(4) $\langle(t = (F_\sigma\ y_1\ \ldots\ y_n) :: E, \theta\rangle \longrightarrow \langle\varphi(E), \varphi \circ \theta\rangle$
provided F is a logic variable and $mksubst(F_\sigma, t, [y_1, \ldots, y_n]) = \varphi$.

Fig. 2. Simplification rules for higher-order pattern unification

a term, it has as arguments a sequence of distinct variables bound by abstractions or distinct constants with label greater than that attached to the logic variable.[2] This is what is known as the higher-order pattern restriction [7, 13] and it has been observed to be satisfied by almost all unification problems that arise in real computations [6]. The solution to such problems can be computed by descending through the structures of terms first in a simplification mode and later in a variable binding mode if needed [10]. The rules in Figure 2 sketch the general form of this process. These rules use lists of equations to capture recursion through term structure. To determine if $\theta \in unify(A, A')$, we initiate the rewriting process with the tuple $\langle A = A' :: nil, \emptyset\rangle$, hoping to reduce it to the form $\langle nil, \theta\rangle$. Notice that rule (2) requires a most general unifier to be computed for two types. This is actually an instance of the well understood first-order unification problem. We also use in this rule the fact that if two terms of identical type have the same constant or bound variable as their heads, then they must have the same number of arguments.

The invocation of $mksubst(F_\sigma, t, [y_1, \ldots, y_n])$ in the last two rules initiates the variable binding phase. This computation is intended to determine a substitution for F_σ and possibly for logic variables appearing in t that make the terms $(F_\sigma\ y_1\ \ldots\ t_n)$ and t identical, if they are in fact unifiable. Towards this end, a traversal is carried out over the structure of t, determining at each subterm what needs to be done with the head symbol if a unifying substitution is to be generated. If this symbol is a constant with label less than or equal to that of F_σ or if it is a variable bound by an abstraction appearing inside t, then it can appear directly in the term to be substituted for F_σ. If it is a constant that has a label larger than that of F_σ or it is a variable bound by an abstraction outside of t, then it may appear in an instance of $(F_\sigma\ y_1\ \ldots\ y_n)$ only if it is in the list $[y_1, \ldots, y_n]$ and in this case the term that F_σ is bound to must carry out a suitable projection. Finally, the head symbol may itself be a logic variable. Suppose that the subterm is $(G_\rho\ z_1\ \ldots\ z_m)$ where G_ρ is a logic variable and the arguments $z_1, \ldots, z_m$ satisfy the pattern restriction. Now, the only situation in which G_ρ may be identical to F_σ with the terms still being unifiable is when the subterm under consideration is all of t. If this is the case, then n must be

[2] We assume that the term is in normal form in making this determination.

identical to m and the the substitution for F_σ should prune away all the arguments for which y_i and z_i do not agree. If G_ρ is distinct from F_σ, we have two cases to consider. In one case, the label of G_ρ may be smaller than or equal to that of F_σ. In this case, it is necessary to "prune" those elements of $z_1, \ldots, z_m$ that do not appear in $y_1, \ldots, y_n$ and a suitable pruning substitution for G_ρ and a corresponding projection for the subterm must be computed. If the label of G_ρ is larger than that of F_σ, it is necessary to replace this variable in the subterm by one that has the same label as F_σ to prevent subsequent instantiations that violate scope restrictions. However, while doing this, the elements of $y_1, \ldots, y_n$ that can legitimately appear in an instantiation of G_ρ and that are not already contained in $z_1, \ldots, z_m$ must be added to the sequence of arguments of the subterm. To realize this correctly, the earlier described pruning substitution for G_ρ must be complemented by a "raising" component.

The above sketch of *mksubst* suffices for the purposes of this paper. We refer the reader interested its detailed presentation to [10]. Relative to that description, we have the following theorem:

Theorem 1. *Let $\mathcal{P}$ be a program and let G be a goal and let $\mathcal{P}'$ and G' be obtained from these by labelling all the constants appearing in them with the number* 0. *Further suppose that all the terms appearing in a derivation rooted at $\mathcal{P}', 0 \vdash G', \theta$ (for an arbitrary θ) satisfy the higher-order pattern restriction. Then there is a derivation of $\mathcal{P}', 0 \vdash G', \varphi$ for some φ if and only if there is a $G' \in \{G\}_t$ such that $\{\mathcal{P}\}_t \vdash G'$ in intuitionistic logic.*

There is a shortcoming in the computation process that we have described in that it "stalls" when it encounters a higher-order unification problem outside the higher-order pattern fragment. There is a simple solution to this that is used in practical systems (e.g. see [14]): the equation that causes the process to stall may be deferred and reexamined later or presented as a qualification on computed answers. The detailed technical development of this approach is orthogonal to the runtime treatment of types and hence we do not discuss it any further.

4 Using Declared Types to Simplify Type Annotations

Types need to be carried into runtime computations only insofar as they affect the course of computation. Towards understanding how this might happen, it is useful to consider the different phases of the interpreter that was presented in the previous section.

In the first phase, characterized by the rules in Figure 1, goals are simplified and a unification computation may be initiated in support of backchaining. Types do not determine the steps in this phase although some bookkeeping work relating to them may have to be done. In particular, the rules [ALL] and [SOME] must attach the type of the quantified variable to the new constant and logic variable introduced by these rules if in fact these types are needed later during execution. An important point to note with these constants and variables, though, is that the same type is shared by every instance and, in terms of checking identity, a simple lookup of the names suffices.

The next phase, defined by the rules in Figure 2, corresponds to the simplification of the top-level fixed structure of terms in the unification process. Types are used in an essential way in one of these rules, specifically in rule (2). In determining the applicability of this rule, it is necessary to match up both the name and the type of the constants or abstracted variables that appear as the heads of the two terms being unified. Observe, however, that if these heads are matching abstracted variables or constants introduced by the [ALL] rule for goals, then the types must already be identical. Thus the checking or unification of types is necessary only for the genuinely polymorphic constants declared at the top-level in the program.

The final phase is the one that determines variable bindings in unification. A closer look at the description we have provided of the computation carried out by *mksubst* reveals the following facts. First, the types of logic variables are neither examined nor refined in the process of constructing bindings. Notice that we do have to check the identities of these variables at certain places but, by virtue of our earlier observation, a simple comparison of names is all that is needed for this. Second, we sometimes have to compare constants (and abstracted variables), but these comparisons are all restricted to the ones that appear as the arguments of the logic variable in the appropriate instance of rule (3) or (4) in Figure 2. The higher-order pattern restriction requires that the constants in this collection have a higher label than the logic variable at the head, implying thereby that the must have been introduced by the use of an [ALL] rule. Hence every instance of any one of these constants must already be known to have the same type. From these observations, it is evident that types are incidental to the variable binding computation.

The above analysis makes clear the fact that the only symbols with which we need to maintain types at run time are the top-level declared constants. A further examination allows us to simplify even this information. In the first instance, the defined type for such a constant provides a skeleton that compile-time type checking ensures every occurrence of the constant shares. The only possible differences between the types of distinct occurrences is in the instantiations of the variables that occur in the skeleton. Thus, the type annotations for each constant can be systematically transformed by a compiler into a (possibly empty) list of type variable instantiations and it is only the simpler types in this list that need to be unified during execution. As a particular example, given the types *(list A)* for *nil* and $A \rightarrow$ *(list A)* $\rightarrow$ *(list A)* for *::*, a compiler can determine that only the bindings for the type variable A need to be stored with instances of these constants. Let us write type annotations as a special first list argument for constants and let us temporarily use an prefix syntax for *::*. Then, by virtue of the present observation, the structure

$$(::\ [int \rightarrow (list\ int) \rightarrow (list\ int)]\ 1\ (nil\ [list\ int]))$$

can be rendered into the form *(:: [int] i (nil [int]))* instead.

The manner in which unification problems are processed actually allows for a further refinement of type annotations. The usage of the rules in Figure 2 begins with an equation between two (predicate) terms that have the same type and each transformation preserves this relationship between the terms in each equation. Thus, at the time when the types of different instances of a constant are being unified in rule (2), their target types are known to be identical. This has the special implication that there is no need to check the bindings for the variables in the type skeleton that also occur in the target type and so these may be eliminated from the annotations. In the case that all the variables in the skeleton type also appear in the target type, i.e., when the constant type satisfies the type preservation property [2], the compiler can conclude that no type annotation needs to be maintained with the constant. This happens to be the case for both *::* and *nil*, for instance, and so all type information can be elided from lists that are implemented using these constants.

We formalize the ideas expressed up to this point in the following fashion. First, we attach with each constant an initial "list of types" argument. This list is empty for the constants introduced by the [ALL] rule and for instances of the other constants it consists of bindings for the variables that appear only in the argument part of their declared types, presented in an order determined by a compiler. This extra argument is simply carried along with the constant when a variable substitution is being constructed. The only real use of it occurs in rule (2) of the simplification phase of unification that is refined into the form shown in Figure 3. The second rule in this collection is needed because constructors of function type can appear without their arguments in programs in our higher-order language. We also note that the types list argument is likely to be empty in most situations and it may be profitable to provide a distinct treatment of this case in an implementation.

(2.1) $\langle (a\ [\tau_1, ..., \tau_k]\ t_1 ... t_n) = (a\ [\sigma_1, ..., \sigma_k]\ s_1 ... s_n) :: E, \theta\rangle$
$\longrightarrow \langle \phi((t_1 = s_1) :: ... :: (t_n = s_n) :: E), \phi \circ \theta\rangle$,
where $n > 0$, and ϕ is a most general unifier for $\{\langle \tau_1, \sigma_1\rangle, \ldots, \langle \tau_k, \sigma_k\rangle\}$,
if a is a constant.

(2.2) $\langle (a\ [\tau_1, ..., \tau_k]) = (a\ [\sigma_1, ..., \sigma_k]) :: E, \theta\rangle \longrightarrow \langle E, \theta\rangle$,
if a is a constant.

(2.3) $\langle (a\ t_1 ... t_n) = (a\ s_1 ... s_n) :: E, \theta\rangle \longrightarrow \langle ((t_1 = s_1) :: ... :: (t_n = s_n) :: E), \theta\rangle$,
is a is a variable bound by an abstraction.

Fig. 3. The refined structure simplification rule

The correctness of the implementation scheme we have described in this section is stated in the following theorem. The proof of this theorem requires, first of all, a specific formal presentation of the compiler function that transforms the types of constants into lists of type variable bindings. A subsequent argument then builds on this definition to establish a correspondence between compile-time type checking and the runtime type unification in rule 2.1 in Figure 3 on the one hand and the unification that is carried out at runtime over the entire type in rule (2) of Figure 2 on the other hand.

Theorem 2. *The modified interpreter described in this section in combination with the scheme for transforming type annotations is sound and complete with respect to the interpreter presented in Section 3.*

The ideas we have described here may be applied to the *append* program. We note that *append* has a type variable appearing in its argument types that does not appear in its target type and the binding for this variable must therefore annotate its occurrences. We have already seen that type annotations can be dropped from *::* and *nil*. Thus, the definition of *append* is transformed into the following:

append [A] nil L L.
(append [A] (X::L1) L2 (X::L3)) :- (append [A] L1 L2 L3).

The query considered in Section 2 correspondingly becomes

$\forall a$ *(append [i] (a::nil) (b::a::nil) (F a)).*

The scheme that we have described is capable also of dealing with the situation where the type preservation property is violated. As an example, consider a representation of heterogenous lists based on the constants *null* of type *lst* and *cons* of type $A \rightarrow lst \rightarrow lst$. The list containing *1* and *"list"* as its elements would then be represented by the following term:

(cons [int] 1 (cons [string] "list" null)).

5 Eliminating Type Annotations for Predicates

Predicate names are constants whose defined types have *o* as their target types. A consequence of this is that the ideas of the previous section do not allow any of the variables that appear in the type of a predicate constant to be dispensed with from the annotation that adorns it. This is unfortunate because in many instances these annotations have no tangible effect on a computation. A particular illustration of this fact is provided by the transformed definition of *append* that we saw towards the end of Section 4. The type variable A that annotates the head of each of these clauses can be unified with any type and hence has no impact on the applicability of the clause to a given query. Actually carrying out its unification with an incoming type will result in extracting a binding that, in the second clause, is passed on to a recursive call of *append*. However, this call will also at most result in the type binding being extracted and passed along without affecting the computation in an observable way. The type annotation for *append* can therefore be dispensed with without adverse effect.

But is there a systematic way for identifying situations in which a type annotation on a predicate constant can be so eliminated? This is the issue we address in this section. The approach we adopt consists of determining which elements of the types list associated with a predicate name could potentially influence the course of a computation. For the others, we can conclude that they will never be needed in an essential way and hence they can be eliminated.

The process of determining the potentially "needed" elements in the types list can be oriented around the clauses defining the predicate constant.[3] We must include in this analysis also the clauses that appear on the lefthand sides of implication goals in the bodies of clauses. If a constant appears as the head of such a clause, we assume every element in its types list is needed: in the model of computation we have described, the values for the type variables that appear in such a clause get fixed when the clause is added to the program and consequently runtime unification with them may determine a binding that influences the subsequent usage of the clause. For a clause that appears at the top-level, our analysis can be more sophisticated. A particular element in the types list for its head predicate is needed if the value in the relevant position in the list associated with the head in that clause is anything other than a variable; unification over this element must be attempted during execution since it has the possibility of failing in this case. Another situation in which the element is needed is if it is a variable that occurs elsewhere in the same types list or in the types lists associated with a non-predicate constant that occurs in the clause. The rationale here is that either the variable will already have a binding that must be tested against an incoming type or a value must be extracted into it that is used later in a unification computation of consequence. A more subtle situation for the variable case is when it occurs in the types list associated with the predicate head of a clause that appears on the left of an implication goal in the body. In this case the binding that is extracted at runtime in the variable has an impact on the applicability of the clause that is added and consequently is a needed one.

The only case that remains to be considered is that where a variable element in the types list for the clause head appears also in the types list associated with a predicate constant in a goal position in the body, either at the top-level or, recursively, in an embedded clause definition. We could, somewhat simplistically, treat such predicate constants also like the other constants. The drawback with this is that the type annotation with the predicate constant appearing in the body may itself be eliminable and then an opportunity for optimization would be missed. We could, of course, determine this neededness information for the body predicate constant first and then use this information in the analysis for the given clause head. As an example of how this might work, suppose that *print* is a predicate of type $A \rightarrow o$ and *printlist* is a predicate of type *(list A)* $\rightarrow o$ and consider the following clauses annotated in the style of Section 4:

print [int] X :- $\{$*code for printing an integer value*$\}$.
print [string] X :- $\{$*code for printing a string value*$\}$.

printlist [C] nil.
printlist [C] (X::L) :- print [C] X, printlist [C] L.

[3] The calculation we describe is sensitive to our being able to fix statically the full set of clauses for a predicate. We obtain this ability here by assuming that the top-level goal does not contain implications. In reality, the module system of λProlog gives assistance in this task. A detailed discussion is beyond the scope of this paper.

In this code, *print* is predicate that is polymorphic in an *ad hoc* way and that makes genuine use of its type "argument." This information can be used to determine that it needs its type adornment and the following analysis exposes the fact that *printlist* must therefore carry its type annotation.[4]

The approach suggested above needs refinement to be applicable to a context where dependencies between definitions can be iterated and even recursive; at present, it doesn't apply directly even the definition of *append*. The solution is to use an iterative, fixed-point computation that has as its starting point the neededness information gathered by initially ignoring predicate constants appearing in goal positions in the body of the clause. In effecting this calculation relative to a given program $\mathcal{P}$, we employ a two-dimensional global boolean array called *needed* whose first index, p, ranges over the set of predicate constants appearing in $\mathcal{P}$ and whose second index, i, is a positive integer that ranges over the length of the types list for p; this array evidently has a variable size along its second dimension. The intention is that if, at the end of the computation, *needed*$[p][i]$ is *false* then the ith element in the types list associated with p does not have an influence on the solution of any goal G from $\mathcal{P}$. We compute the value of this array by initially setting all the elements of *needed* to *false* and then calling the procedure *find_needed* defined in Figure 4 on the program $\mathcal{P}$.

The invocation of *find_needed* on any program $\mathcal{P}$ must clearly terminate. The correctness of the procedure is then the content of the following lemma.

Lemma 1. *Let p be a predicate constant defined in $\mathcal{P}$. Further, let it be the case that when find_needed($\mathcal{P}$) terminates, needed[p][i] is set to false. Then the ith element in the types list associated with p has no impact on the solvability of any goal G from $\mathcal{P}$.*

Proof. Only a sketch is provided. Suppose that the specific value of a component of the types list of a predicate constant p has a relevance to a some computation. Then it must become relevant at a specific point in the backchaining sequence. An induction on the distance of the relevant call of p from this point in the sequence shows that *needed[p][i]* must have been set to true by *find_needed*: the base case is accounted for by the initialization code and the inductive case is handled by the fact that the iteration concludes only when a fixed point is reached.

The lemma leads naturally to the following theorem:

Theorem 3. *Let $\mathcal{P}$ and G be a program and a goal that is annotated in style described at the end of Section 4. Let $\overline{\mathcal{P}}$ and $\overline{G}$ be the program and goal that result from $\mathcal{P}$ and G by eliminating those components from the types lists of predicates that are found not to be needed by the invocation of find_needed($\mathcal{P}$). Then G succeeds from $\mathcal{P}$ if and only if $\overline{G}$ succeeds from $\overline{\mathcal{P}}$ using the interpreter described in Section 4.*

[4] This example vividly illustrates the problem with interpreting the conditions described in [1] and [2] as applicable on a "per clause" and "per constant" basis. Using them in this way, we would drop the type annotation with *print_list* and therefore not be able to pass this information on to *print* where it is genuinely needed.

```
procedure $find\_needed(\mathcal{P})$ {
  $init\_needed(\mathcal{P})$;
  repeat
    for each top-level non-atomic clause $C$ in $\mathcal{P}$ {$process\_clause(C)$;}
  until (the value of $needed$ does not change)
}

procedure $init\_needed(\mathcal{P})$ {
  for every embedded clause $C$ in $\mathcal{P}$ with $(p\ [\tau_1, ..., \tau_k]\ t_1\ ...\ t_n)$ as head
    for $1 \leq i \leq k$ {$needed[p][i] = true$};
  for every top-level clause $C$ in $\mathcal{P}$ with $(p\ [\tau_1, ..., \tau_k]\ t_1\ ...\ t_n)$ as head
    for $1 \leq i \leq k$
      if $\tau_i$ is not a type variable {$needed[p][i] = true$;}
      else {
        if (($\tau_i$ occurs in $\tau_j$ for some $j$ such that $1 \leq j \leq k$ and $i \neq j$) or
            ($\tau_i$ occurs in the types list of a non-predicate constant in $C$) or
            ($\tau_i$ occurs in the types list of a predicate constant appearing
             as the head of an embedded clause in the body of $C$))
            $needed[p][i] = true$;
      }
}

procedure $process\_clause(C)$ {
  let $C$ be of the form $(p\ [\tau_1, \ldots, \tau_k]\ t_1\ \ldots\ t_n)$ :- $G$
    for $1 \leq i \leq k$
      if $needed[p][i]$ is $false$ then {$needed[p][i] = process\_body(G, \tau_i)$};
}


function $process\_body(G, \tau)$ : boolean {
  if $G$ is
    $\forall G'$ , $\exists G'$ : return $process\_body(G', \tau)$;
    $G_1 \wedge G_2$: return ($process\_body(G_1, \tau)$ or $process\_body(G_2, \tau)$);
    $D \supset G$: return ($process\_body(G, \tau)$ or $process\_embedded\_clause(D, \tau)$);
    atomic and of the form $(q\ [\sigma_1, ..., \sigma_l]\ s_1\ ...s_m)$:
      if $\tau$ occurs in $\sigma_i$ for some $i$ such that $1 \leq i \leq l$ and $needed[q][i]$ is $true$
      then return $true$;
      else return $false$;
}


function $process\_embedded\_body(D, \tau)$ : boolean {
  if $D$ is
    $\forall D_1$ : return $process\_embedded\_body(D_1)$;
    $G \supset A$: return $process\_body(G, \tau))$;
    atomic: return $false$;
}
```

Fig. 4. Determining if a predicate type argument is needed

Using this theorem and *find_needed*, the type annotation for *append* can be eliminated and the definition of this predicate can be reduced to essentially the untyped form. In general, if every clause is type general in the sense of [2], then types can be eliminated entirely from runtime computations. Note, however, that we permit programs that do not satisfy this property and that our ideas can be useful in reducing type annotations even with such programs.

6 Conclusion

A polymorphically typed higher-order logic programming language like λProlog requires type information to be carried into computations. We have described in this paper ways in which the amount of information that must be available and manipulated at runtime can be significantly reduced. A critical part of our approach is a shift from using a full higher-order unification procedure to one based on higher-order patterns. There can be some differences in the end results of computations as a result of this shift but, in most cases, the changes are actually for the better in that more precise answers are produced. The modified model also facilitates a static analysis of the dynamic effects of types that eventually lies at the heart of our approach for eliding them in programs.

The ideas we have described here need extension in one respect to be actually applicable to λProlog. In this language, predicate constants can in fact appear within terms. When they appear in such contexts, they have to be treated like other (non-predicate) constants and, under the present scheme, must carry binding for their type variables. However, even in this situation, the ideas in Section 5 can be applied to the extensional uses of predicate constants. Moreover, by exploiting visibility properties of constants emanating from the modules language of λProlog, we can profitably lift the kind of analysis that we have described in Section 5 for predicate constants that appear extensionally to constants that appear within terms. As a particular case, then, the reach of these ideas can also be extended to constants that appear both intensionally and extensionally.

The work that we have described here is being utilized in a new implementation of λProlog. They already have an impact in yielding an abstract machine for the language that is considerably simpler than the one underlying the *Teyjus* system [11]. We expect in the future to be able to compare the performance of the two systems and to isolate the efficiency benefits of the reduced type processing that are supported by the ideas in this paper.

Acknowledgements

Support for this work has been provided by the NSF through the grant numbered CCR-0429572 and by the Digital Technology Center and the Department of Computer Science and Engineering at the University of Minnesota.

References

1. M. Hanus. Horn clause programs with polymorphic types: Semantics and resolution. In J. Diaz and F. Orejas, editors, *TAPSOFT 89*, pages 225–240. Springer-Verlag, 1989. Lecture Notes in Computer Science Vol 352.
2. M. Hanus. Polymorphic higher-order programming in Prolog. In G. Levi and M. Martelli, editors, *Proceedings of the Sixth International Logic Programming Conference*, pages 382–398. MIT Press, 1989.
3. G. Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1:27–57, 1975.
4. K. Kwon, G. Nadathur, and D.S. Wilson. Implementing polymorphic typing in a logic programming language. *Computer Languages*, 20(1):25–42, 1994.
5. T.K. Lakshman and U.S. Reddy. Typed Prolog: A semantic reconstruction of the Mycroft-O'Keefe type system. In V. Saraswat and K. Ueda, editors, *Proceedings of the International Logic Programming Symposium*, pages 202–217. MIT Press, 1991.
6. S. Michaylov and F. Pfenning. An empirical study of the runtime behavior of higher-order logic programs. In *Conference Record of the Workshop on the λProlog Programming Language*, Philadelphia, July-August 1992.
7. D. Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1(4):497–536, 1991.
8. D. Miller, G. Nadathur, F. Pfenning, and A. Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.
9. A. Mycroft and R. A. O'Keefe. A polymorphic type system for Prolog. *Artificial Intelligence*, 23:295–307, 1984.
10. G. Nadathur and N. Linnell. Practical higher-order pattern unification with on-the-fly raising. Technical Report 2005/2, Digital Technology Center, April 2005. To appear in the Proceedings of ICLP'05.
11. G. Nadathur and D.J. Mitchell. System description: Teyjus—a compiler and abstract machine based implementation of λProlog. In H. Ganzinger, editor, *Automated Deduction–CADE-16*, number 1632 in Lecture Notes in Artificial Intelligence, pages 287–291. Springer-Verlag, July 1999.
12. G. Nadathur and F. Pfenning. The type system of a higher-order logic programming language. In F. Pfenning, editor, *Types in Logic Programming*, pages 245–283. MIT Press, 1992.
13. T. Nipkow. Functional unification of higher-order patterns. In *Eighth Annual IEEE Symposium on Logic in Computer Science*, pages 64–74. IEEE Computer Society Press, June 1993.
14. F. Pfenning. Elf: A language for logic definition and verified metaprogramming. In *Fourth Annual Symposium on Logic in Computer Science*, pages 313–322. IEEE Computer Society Press, June 1989.
15. D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A type-directed optimizing compiler for ML. In *Proc. ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 181–192, 1996.