A Treatment of Higher-Order Features in Logic Programming

Gopalan Nadathur
Department of Computer Science and Engineering
University of Minnesota
4-192 EE/CS Building, 200 Union Street S.E.
Minneapolis, MN 55455
gopalan@cs.umn.edu

Abstract

The logic programming paradigm provides the basis for a new intensional view of higher-order notions. This view is realized primarily by employing the terms of a typed lambda calculus as representational devices and by using a richer form of unification for probing their structures. These additions have important meta-programming applications but they also pose non-trivial implementation problems. One issue concerns the machine representation of lambda terms suitable to their intended use: an adequate encoding must facilitate comparison operations over terms in addition to supporting the usual reduction computation. Another aspect relates to the treatment of a unification operation that has a branching character and that sometimes calls for the delaying of the solution of unification problems. A final issue concerns the execution of goals whose structures becomes apparent only in the course of computation. These various problems are exposed in this paper and solutions to them are described. A satisfactory representation for lambda terms is developed by exploiting the nameless notation of de Bruijn as well as explicit encodings of substitutions. Special mechanisms are molded into the structure of traditional Prolog implementations to support branching in unification and carrying of unification problems over other computation steps; a premium is placed in this context on exploiting determinism and on emulating usual first-order behaviour. An extended compilation model is presented that treats higher-order unification and also handles dynamically emergent goals. The ideas described here have been employed in the *Teyjus* implementation of the λProlog language, a fact that is used to obtain a preliminary assessment of their efficacy.

Keywords: lambda calculus, intensional higher-order programming, higher-order unification, abstract machine, compilation.

1 Introduction

Customary acquaintance with higher-order notions in programming relates to the imperative or functional programming paradigms. Within these frameworks, functions are equated with the methods for computing that are embodied in procedures. Higher-orderness then consists of the ability to objectify such functions and, thereby, to embed them in data or pass them as arguments to other functions. Many interesting applications have been found for such capabilities. However, all of these are dependent on a uniform *extensional* view of functions. In particular, the only observable aspect of such objects is their ability to transform given arguments into result values.

Logic programming has the potential for supporting a different and, in some senses, more sophisticated understanding of higher-order notions [36]. Functions are used within this paradigm as a means for constructing descriptions of objects. Such descriptions can be examined by means of unification, an operation that is useful in the analysis of *intensions*. Traditional logic programming languages manifest a weak exploitation of this capability because they permit only individual, non-function, objects as values. However, it is possible to support the probing of function structure in genuinely higher-order ways by introducing a mechanism such as the terms of a lambda calculus for encoding function objects and by complementing this with richer notions of variables and unification. The usual form of higher-order programming can be realized simply by using the ability to represent function valued objects and the extensional interpretation built into logic programming of one kind of function, namely, predicates. The richer intensional view of functions offers, in addition, many possibilities that have not been systematically supported by any previous programming paradigm. To consider one important direction, the ability to use lambda terms as representational devices lends itself well to an abstract view of syntax that treats binding notions explicitly [27, 41], leading thereby to many novel metalanguage applications for logic programming (*e.g.*, see [5, 15, 17, 18, 40]).

While there is considerable application potential for higher-order features in logic programming, the addition of such features also raises significant implementation problems. One category of problems arises from the use that is made of the terms of a lambda calculus essentially as data structures. This is a truly novel role for such terms in programming, and a representation must be developed for them that supports their use in this capacity. A satisfactory representation should permit the examination of term structure and must facilitate the comparison of terms in a situation where the particular names of bound variables are unimportant, in addition to efficiently supporting the usual reduction operation on terms. Another class of problems relates to the fact that the unification computation on lambda terms, known as higher-order unification [20], possesses characteristics that are distinct from those of the customary first-order unification. In particular, performing this operation may involve a branching search and it may also be necessary to temporarily 'suspend' the computation before a unifier is found. Suitable dynamic support must be described for both facets. A third aspect that needs special consideration is the mixing of intensional and extensional views of predicate objects. There is a distinction between examining the structure of an object and using this structure to determine the invocation of code. A satisfactory method must be provided for realizing the switch between these roles. Finally, any machinery that is designed for supporting the new features must be interwoven into the run-time devices and approaches to compilation that are commonly used in logic programming implementation. The proper combination of all these mechanisms in one system is itself a non-trivial issue.

We consider all these problems in this paper and we develop methods for addressing them. For the sake of concreteness, we describe our new implementation ideas within the framework of the Warren Abstract Machine (WAM) [43], a popular vehicle for realizing logic programming languages. One of our contributions relates to the representation of lambda terms. We carefully identify the different issues that become relevant where these terms are used intensionally and we develop an encoding for them that utilizes mechanisms for eliminating bound variable names [9] and for capturing substitutions in terms [39] towards addressing these issues. We also outline a low-level realization of such an encoding and we discuss the integration of operations on these terms into an abstract machine structure. Another contribution is the development of machinery for supporting the special needs of higher-order unification. In this direction we, first of all, describe an explicit encoding of unification problems that exploits the manner in which these evolve to foster sharing in their representation. We also propose mechanisms that are suitable for realizing branching in unification through a depth-first search regimen with the possibility of backtracking. Finally, recognizing that branching search is, in general, computationally expensive, we describe a processing structure that facilitates the application of special deterministic steps and that delays the consideration of branching until after such steps. Using this approach it is possible to treat first-order unification almost exactly as it would be treated in a Prolog implementation, a facet recognized to be important even in higher-order programs [25]. The final contribution of this paper relates to compilation. We propose enhancements to the structure of the WAM and modifications to its instruction set that together realize a compiled execution of programs in a higher-order language. We also outline in this context a treatment of the transition between intensional and extensional roles of predicates.

The ideas that we develop in this paper have a special practical relevance: they are useful to the implementation of the logic programming language λProlog [34]. This language actually embodies two extensions to a Prolog-like language in addition to the higher-order features considered here. In one direction, it makes richer use of logical connectives and quantifiers to introduce notions of scoping [29]. In another direction, it includes a polymorphic typing regimen [38]. Both aspects raise new questions for implementation that we have addressed elsewhere [22, 33]. The machinery that we describe here blends well with these other mechanisms and all our ideas have, in fact, been amalgamated in a new implementation of λProlog called *Teyjus* [37].

The rest of this paper is structured as follows. In the next section we identify a logic programming language that embodies the higher-order features that are presently of interest and we characterize computation in this language. In Section 3, we refine the description of computation into one that provides the basis for implementation by outlining the structure of the higher-order unification operation and using this to develop an abstract interpreter for the language. The remainder of this paper concerns a low-level realization of this interpreter. In Section 4, we discuss issues relevant to the representation of lambda terms and distill from this an encoding for them that can be used in an actual implementation. The following section integrates our term representation into an overall computational model and proposes new machinery for the realization of higher-order unification. Section 6 makes explicit an extended abstract machine structure and considers the compilation of first-order like unification as well as the treatment of higher-order aspects relative to this machine. Section 7 discusses related work and concludes the paper.

2 A Higher-Order Language

The logical language whose implementation we consider in this paper is an analogue within Church's Simple Theory of Types [10] of the Horn clause language that underlies Prolog. Church's logic is one that builds on a typed lambda calculus. In the interpretation we use here, the types are constructed from given sets of sorts and type constructors, each element of the latter set being attributed a specific arity. The set of sorts initially contains o, the type of propositions, and others such as *int*, *real*, *etc*, with obvious interpretations. We also assume the availability of at least the unary list type constructor *list*. Both these sets must be augmentable dynamically in a programming situation, a fact that we will utilize implicitly. The full collection of types is the smallest set satisfying the following properties: (i) each sort is a type, (ii) if $\alpha_1, \ldots, \alpha_n$ are types and $c \in \mathcal{C}$ is of arity n, then $(c\ \alpha_1\ \ldots\ \alpha_n)$ is a type, and (iii) if α and β are types, then so is $(\alpha \to \beta)$. A function type is one whose top-level structure has the form $(\alpha \to \beta)$. All other types are considered to be atomic. We minimize the use of parentheses by assuming that the application of a type constructor has highest priority and that $\to$ is right associative. The latter convention allows any function type to be written in the form $\alpha_1 \to \ldots \to \alpha_n \to \beta$ where β is an atomic type. The target type of such a type is β and $\alpha_1, \ldots, \alpha_n$ are its argument types. This notation and terminology is extended to atomic types by permitting the argument types to be an empty sequence.

Starting from typed collections of constants and variables, the terms in the language are identified together with their types via the following rules: (i) each constant and variable of type α is a term of type α, (ii) if x is a variable of type α and t is a term of type β, then $(\lambda x\, t)$ is a term of type $\alpha \to \beta$ and is called an abstraction that binds x and has scope t, and (iii) if t_1 and t_2 are terms of type $\alpha \to \beta$ and α respectively, then $(t_1\ t_2)$ is a term of type β and is called an application of t_1 to t_2. We reduce the use of parentheses in writing terms by assuming that application is left associative and that an abstraction has as its scope the largest well-formed term to the right of the variable it binds.

The constants in the language are partitioned into *non-logical* and *logical* ones. The former set contains an initial collection of elements such as those representing the integers and is augmentable in some manner that we, once again, leave implicit. The set of logical constants consists of the symbols $\top$ of type o denoting the tautologous proposition, $\neg$ of type $o \to o$ denoting negation, $\wedge$, $\vee$ and $\supset$ of type $o \to o \to o$ denoting conjunction, disjunction and implication, respectively, and, for each α, Σ_α and Π_α of type $(\alpha \to o) \to o$. The last two 'families' of constants represent generalized existential and universal quantifiers: formulas usually written as $\exists x\, P$ and $\forall x\, P$ are rendered in this logic as $\Pi_\alpha\ \lambda x\, P$ and $\Sigma_\alpha\ \lambda x\, P$ for an appropriate type α. We will, in fact, use the former as abbreviations of the latter. Although type subscripts are strictly necessary with Π and Σ, we will omit these when their identity is obvious or irrelevant to the discussion at hand. We will also adopt the customary infix notation for the application of $\wedge$, $\vee$ and $\supset$ to two arguments in succession.

We assume the usual notions of free and bound variables and of subterms of a term. Equality between terms incorporates the rules of lambda conversion. Let us say that a term s is free for the variable x in the term t if x does not appear free in t in the scope of an abstraction that binds a free variable of s. Further, let $t[x := s]$ denote the result of replacing all the free occurrences of x in t by s. The lambda conversion rules are then the following:

1. (α-conversion) Replacing a subterm of the form $\lambda x\, t$ in a given term by $\lambda y\, (t[x := y])$, provided y is free for x in t.
2. (β-conversion) Replacing a subterm of the form $(\lambda x\, t_1)\ t_2$ in a given term by $t_1[x := t_2]$ or vice versa, provided t_2 is free for x in t_1.
3. (η-conversion) Replacing a subterm of the form t in a given term by $\lambda x\, t\ x$ and vice versa, provided t is of type $\alpha \to \beta$, x is of type α and x does not appear free in t.

Two terms are considered to be equal if one can be obtained from the other by using a sequence of these rules. In determining such equality, it is often necessary to consider directed applications of these conversion rules. Of particular importance is an oriented form of the β-conversion rule that is made precise as follows. First, we identify a term of the form $(\lambda x\, t_1)\ t_2$ as a *β-redex*; in the sequel, we shall also refer to t_1 as the body of this redex and to t_2 as its argument. Now, the condition permitting the replacement of a subterm of this kind as per the β-conversion rule may not be satisfied in general, but this can be corrected by using a sequence of α-conversion steps. We call such a sequence followed by the desired application of the β-conversion rule a β-contraction.

We will need to consider the idea of unifying two lambda terms. The interest here is in substituting terms of matching types for free variables so that the two terms become equal. This substitution operation must be performed with care to ensure that free variables in the substituted terms do not get accidentally bound in the result. A correct characterization of this operation can, in fact, be provided using equality modulo the lambda conversion rules. Thus, suppose that, for $1 \leq i \leq n$, t_i and x_i are a term and a variable of identical type. Then, the set $\{\langle x_i, t_i\rangle | 1 \leq i \leq n\}$ represents a substitution and the application of this substitution to t is equal to the term $(\lambda x_1 \ldots \lambda x_n\, t)\ t_1\ \ldots\ t_n$.

A central part of generalizing Horn clauses to higher-order logic is describing a suitable notion of atomic formulas. Towards this end, we first identify the class of *positive* terms as those lambda terms in which the only logical constants that appear are $\wedge$, $\vee$ and Σ_α. Our atomic formulas or atoms are then all the terms of type o of the form $P\ t_1\ \ldots\ t_n$ where P is a (predicate) variable or non-logical constant and, for $1 \leq i \leq n$, each t_i is a positive term. Such a formula is said to be *rigid* just in case P, its predicate head, is a non-logical constant and is said to be *flexible* otherwise. We denote arbitrary atoms by A and rigid ones by A_r below. *Goal formulas* or simply *goals* are then the propositional terms that are denoted by G in the syntax rule

$$G ::= \top \mid A \mid G \wedge G \mid G \vee G \mid \exists x\, G.$$

These formulas are higher-order versions of queries or goals in Prolog; notice, in particular, that the arguments of atomic goals are lambda terms as opposed to first-order terms and predicate and function variables are permitted in goals. A higher-order Horn clause or program clause is the universal closure of a term of the form A_r or $G \supset A_r$. Program clauses are intended to be interpreted in a computational setting as (partial) definitions of procedures and from this perspective the restriction to rigid atoms is well-motivated: such an interpretation is meaningful only if the 'procedure' has a definite name.

A set of higher-order program clauses constitute a program in our logic programming language. Computation is engendered, as usual, by a goal formula being presented relative to a given program. Such a goal formula typically has existential quantifiers at its head and the programming task is to find instantiations for the quantified variables that permit the resulting goal to be solved from the program.

Example 2.1. Let i be a sort representing individuals and let the set of nonlogical constants contain the following:

nil	of type $list\ i$
$::$	of type $i \to list\ i \to list\ i$, and
mapfun	of type $list\ i \to (i \to i) \to list\ i \to o$

Further, assume that $::$ can be written as an infix, right associative operator.[1] Then the clauses

$\forall f\,(mapfun\ nil\ f\ nil)$ and
$\forall x\, \forall f\, \forall l_1\, \forall l_2\,(mapfun\ l_1\ f\ l_2) \supset (mapfun\ (x\ ::\ l_1)\ f\ ((f\ x)\ ::\ l_2))$

constitute a program.[2] Adopting Prolog's suggestive manner of writing implications, its convention of making quantifiers implicit by using names beginning with uppercase letters for quantified variables and its method for depicting each program clause, this program can be rendered into the following 'friendlier' syntax:

$mapfun\ nil\ F\ nil.$
$mapfun\ (X :: L1)\ F\ ((F\ X) :: L2)$ `:-` $mapfun\ L1\ F\ L2.$

Letting g be a constant of type $i \to i \to i$ and a and b be constants of type i, the formula

$\exists l\ mapfun\ (a :: b :: nil)\ (\lambda x\, g\ a\ x)\ l$

constitutes a query. Using Prolog's conventions for making quantifiers implicit, this query may be rewritten as

[1] The λProlog language contains devices for identifying particular constants as operators.

[2] The omission of types with the quantifiers in these clauses illustrates the convention alluded to earlier. In this instance, the type of *mapfun* uniquely determines the types of the quantified variables.

$$mapfun\ (a :: b :: nil)\ (\lambda x\, g\ a\ x)\ L.$$

There is exactly one solution to this query, this being given by the 'answer' substitution

$$\{\langle L, (g\ a\ a) :: (g\ a\ b) :: nil\rangle\}.$$

Notice that generating this answer substitution requires, amongst other things, the application of a lambda term to two different arguments and the subsequent reduction of these terms to normal form. An alternative query is the following:

$$mapfun\ (a :: b :: nil)\ F\ ((g\ a\ a) :: (g\ a\ b) :: nil).$$

This query also has a unique solution, this being the substitution

$$\{\langle F, \lambda x\, g\ a\ x\rangle\}$$

Computing this answer involves unifying two pairs of terms containing a function variable, these being $F\ a$ and $g\ a\ a$ on the one hand and $F\ b$ and $g\ a\ b$ on the other. We discuss in the next section a process by which such higher-order unification problems may be solved. For the moment, we note that the first of the pairs of terms has four distinct unifiers given by the following substitutions for F:

$$\{\langle F, \lambda x\, g\ a\ a\rangle\},\ \{\langle F, \lambda x\, g\ x\ a\rangle\},\ \{\langle F, \lambda x\, g\ x\ x\rangle\},\ \text{and}\ \{\langle F, \lambda x\, g\ a\ x\rangle\}.$$

If the two pairs of terms in question are unified sequentially and any but the last solution is chosen initially for the first pair, then it will be necessary to backtrack to find a solution for the composite problem. □

We shall employ the conventions used for depicting formulas in the above example freely in the rest of the paper. The predicate *mapfun* considered in this example relates a function and two lists just in case the second list is obtained by applying the function to each element of the first list. The notion of function application is, however, relatively weak, being given by reduction in a typed lambda calculus with no interpreted constants. A stronger form of function application, one that invokes the ability of solving goals in the underlying language, can be realized by using a predicate version of *mapfun* as described in the following example.

Example 2.2. In addition to the constants and types of Example 2.1, assume that *mappred* is a constant of type $list\ i \to (i \to i \to o) \to list\ i \to o$. Then, using the Prolog convention of depicting conjunctions by commas, the following clauses correspond to a program:

$$mappred\ nil\ P\ nil.$$
$$mappred\ (X :: L1)\ P\ (Y :: L2)\ \texttt{:-}\ (P\ X\ Y), (mappred\ L1\ P\ L2).$$

Let *bob*, *john*, *mary*, *sue*, *dick* and *kate* be constants of type i and let *parent* be a constant of type $i \to i \to o$. Then the following additional clauses define a 'parent' relationship between different individuals:

$$
\begin{array}{l}
parent\ bob\ john.\\
parent\ john\ mary.\\
parent\ sue\ dick.\\
parent\ dick\ kate.
\end{array}
$$

In this context, the following term constitutes a query:

$$mappred\ (bob :: sue :: nil)\ parent\ L.$$

The sole answer to this query is the substitution

$$\{\langle L, john :: dick :: nil\rangle\}.$$

In solving this query, two new goals of the form $(parent\ bob\ Y1)$ and $(parent\ sue\ Y2)$ will have to be dynamically formed and solved. Another example of a query is

$$mappred\ (bob :: sue :: nil)\ (\lambda x\, \lambda y\, \exists z\, (parent\ x\ z) \wedge (parent\ z\ y))\ L.$$

This goal asks for the grandparents of *bob* and *sue* and has as its solution the substitution

$$\{\langle L, mary :: kate :: nil\rangle\}.$$

Finding this answer requires two new goals with complex structures—each with an embedded conjunction and existential quantifier—to be constructed at runtime and then solved. □

Example 2.2 motivates the particular structure chosen for atomic formulas in the definition of our higher-order logic programming language. Logical constants that appear in the arguments of predicate expressions can become top-level symbols in a goal constructed at runtime. These constants must, therefore, be limited to ones that can legitimately appear in such a position, a requirement that is achieved by the restriction to positive terms. In a different direction, in contrasting this example with Example 2.1, a question that arises is whether or not the *mappred* predicate can be run in 'reverse'. For example, is the query

$$mappred\ (bob :: sue :: nil)\ P\ (john :: dick :: nil)$$

computationally meaningful? It is tempting to decide that it is and that a suitable answer to it is the substitution $\{\langle P, parent\rangle\}$. However, a little thought reveals that there are too many relations that are true of *bob* and *john* on the one hand and *sue* and *dick* on the other and so this query may be deemed to be an ill-formed one. An alternative viewpoint is to note that there is one substitution for P that subsumes all others in a logical sense, this being $\{\langle P, \lambda x\, \lambda y\, \top\rangle\}$ and to treat this as the only legitimate answer to the posed query.

We have relied thus far on an intuitive understanding of what it means to solve a goal. This understanding can be made logically precise by equating it with the notion of provability in classical logic, an aspect that is explored at length in [35]. At an operational level, this sanctions a recipe for solving a *closed* goal from a program $\mathcal{P}$ that is based on the structure of the goal:

1. Solve $G_1 \wedge G_2$ by solving both G_1 and G_2.
2. Solve $G_1 \vee G_2$ by solving one of G_1 and G_2.
3. Solve $\exists x\, G$ by solving $G[x := t]$ for some closed positive term t.
4. Solve a rigid atom A_r by either
   (a) determining that it is equal to a ground instance of a clause in $\mathcal{P}$, or
   (b) by finding a ground instance $G \supset A'_r$ of a clause in $\mathcal{P}$ such that A_r and A'_r are equal and then solving G.

In this description, a ground instance of a program clause is generated by substituting closed positive terms for the universally quantified variables in the clauses. We observe that this recipe does not include methods for solving $\top$ or flexible atoms. The former kind of goal is solved trivially. Goals of the latter kind do not arise if only closed queries are considered.

While the recipe described above clarifies the operational semantics of our language, it is inadequate as a basis for implementation. One problem with it is that it assumes an oracle for picking a suitable instantiation of an existentially quantified goal. Another problem is that it is nondeterministic: choices have to be made within it concerning the disjunct of a disjunctive goal that is to be solved and the clause that is to be used to solve an atomic formula. These problems arise in the context of a first-order language as well and it is useful to understand the approaches used in that situation for dealing with them. Existential goals are treated there by delaying the choice of actual instantiations till such time that information is available for making the 'right' choices. In particular, a goal of the form $\exists x\, G$ is transformed into one of the form $G[x := X]$ where X is a new (free) variable that may be instantiated in the course of computation. Actual instantiations for such variables are determined at the time of solving atomic goals. Given the atomic goal A, we look for a program clause of the form $\forall y_1 \ldots \forall y_n\, A'$ or $\forall y_1 \ldots \forall y_n\, (G' \supset A')$ that is such that A unifies with the formula that results from A' by replacing the universally quantified variables with new variables. If a clause of this kind is found, then, depending on its form, either the atomic goal succeeds immediately or the next task becomes that of solving the resulting instance of G'. With regard to nondeterminism, the usual solution is to make a choice in a predetermined manner and to reconsider this choice in case of subsequent failure. In particular, disjunctive goals are considered in a left-to-right order and program clauses are used in the order of presentation. It turns out that the treatment of the logical connectives, the sequencing through program clauses and much of the unification involved in this process can actually be compiled and this is, in fact, what is done within machine models such as the WAM.

The ideas discussed above have obvious applicability in the implementation of our language as well. However, their precise deployment must take into account the higher-order nature of this language. A detailed exposition of the new problems posed by this aspect and an integration of their treatment into the basic framework described above is the subject of the rest of this paper.

Before concluding this section, we comment briefly on the rigidity of the typing regimen used in our language. The predicates *mapfun* and *mappred* as we have defined them here are, for instance, restricted to apply to lists of individuals and cannot be used with lists of integers, lists of lists or lists of function objects. This inflexibility can be alleviated by injecting a form of polymorphism through the use of type variables. Thus, with an appropriate change to the underlying typing scheme, *mapfun* may have been defined to be of type $list\ A \rightarrow (A \rightarrow B) \rightarrow list\ B \rightarrow o$, where A and B can be instantiated by arbitrary types. A polymorphism of this kind is, in reality, supported by λProlog. However, we elide this polymorphism here because it poses additional implementation problems that we presently do not wish to consider. For the interested reader, these problems are discussed for a first-order language in [22]. The solutions provided therein are entirely compatible with the implementation methods we develop here for our simply typed language.

3 An Abstract Interpreter

To provide a framework for actual implementation, we refine the recipe for solving goal formulas described in the last section by introducing free variables in formulas and by using unification in conjunction with these for determining suitable instantiations of existentially quantified goals. Constructing such a procedure requires a more detailed understanding of the higher-order unification problem, a matter that we first discuss.

In the general case, a higher-order unification problem is given by a finite collection of pairs of terms in which the two terms in each pair have identical types. Such a set is referred to as a *disagreement set*. A solution to, or a unifier for, a unification problem is a substitution θ that makes the two terms in each pair equal when it is applied to them.

The most widely used approach to solving higher-order unification problems is the one due to Huet [20]. This method relies on a form for terms that is referred to as a head normal form. A term is in such a form if it has the structure $\lambda x_1 \ldots \lambda x_n\,(A\ t_1\ \ldots\ t_m)$ where A is either a constant or a variable. Given such a term, A is called its *head*, the abstractions at the front of the term are collectively called its *binder*, $t_1, \ldots t_m$ are called its arguments, $(A\ t_1\ \ldots\ t_m)$ is called its body and the term is said to be *rigid* if A is a constant or an element of $\{x_1, \ldots, x_n\}$, and *flexible* otherwise.[3] Every term in our typed language can be transformed into such a form modulo the lambda conversion rules [4]. Moreover, the results of applying a substitution to a term and to any one of its head normal forms are equal under these rules. Thus, we may restrict our attention to terms in such a form as we henceforth do.

Huet's unification procedure consists of the repetitive use of two phases for transforming a given disagreement set into a form for which it can be decided no unifiers exist or for which unifiability is evident. The first of these phases is akin to the term simplification that is an intrinsic part of first-order unification. Consider two head normal forms that are of the same type. The binders of these terms may be distinct both in the choice of variable names

[3]The flexible and rigid terminology introduced here is an extension of that described for atomic goals.

and in length at the outset, but these can be arranged to be identical through the use of the α- and η- conversion rules. We may therefore assume that the terms in question are in fact of the form $\lambda x_1 \ldots \lambda x_n\,(A_1\ s_1\ \ldots\ s_i)$ and $\lambda x_1\ \ldots\ \lambda x_n\,(A_2\ r_1\ \ldots\ r_j)$ respectively. Now, if both terms are rigid, it can be seen that they are unifiable only if A_1 and A_2 are identical and, in this case, they have the same unifiers as does the set

$$\{\langle \lambda x_1 \ldots \lambda x_n\, s_1, \lambda x_1 \ldots \lambda x_n\, r_1\rangle, \ldots, \langle \lambda x_1 \ldots \lambda x_n\, s_i, \lambda x_1 \ldots \lambda x_n\, r_i\rangle\};$$

note that the identity of types ensure that $i = j$ if $A_1 = A_2$. Thus, given an arbitrary disagreement set, this observation can be used either to conclude that it has no unifiers or to reduce it to another disagreement set with the same unifiers and in which each pair has at least one flexible term. We assume below that this kind of simplification is carried out by a function called *SIMPL* that returns a distinguished value **F** in the case that it detects the impossibility of unification.

One of the possibilities for the value returned by *SIMPL* is that it is a disagreement set that has only 'flexible-flexible' pairs. A set of this kind is known to be unifiable but, in the case that it is non-empty, a complete search for its unifiers can be unconstrained [20]. The best strategy for these sets is therefore to treat them as constraints on any further processing or, if computation is at an end, to present them as such on computed answers.

The second phase in unification becomes relevant when *SIMPL* returns a set that has at least one 'flexible-rigid' pair. A substitution may be posited for reducing the difference between the terms in the pair in this case. Two kinds of elementary substitutions completely cover all the possible ways of doing this. The first makes the head of the flexible term 'imitate' that of the rigid term, and the second 'projects' it onto one of the arguments in the hope that the head of the resulting term may be made identical to the rigid one. In particular, let t_1 be the flexible term with F as its head and let t_2 be the rigid term with c as its head. Further, let the types of F and c be $\alpha_1 \to \cdots \to \alpha_k \to \beta$ and $\gamma_1 \to \cdots \to \gamma_j \to \beta$ respectively, where β is an atomic type. Then

1. the *imitation substitution* is defined only when c is a constant and is

   $$\{\langle F, \lambda w_1 \ldots \lambda w_k\,(c\ (H_1\ w_1\ \ldots\ w_k)\ \ldots\ (H_j\ w_1\ \ldots\ w_k))\rangle\},$$

   assuming that $H_1, \ldots, H_j$ are new, and hence free, variables of appropriate types, and

2. for $1 \leq i \leq k$, the i^{th} *projection substitution* is defined only when α_i is of the form $\beta_1 \to \cdots \to \beta_l \to \beta$ and is

   $$\{\langle F, \lambda w_1 \ldots \lambda w_k\,(w_i\ (H_1\ w_1\ \ldots\ w_k)\ \ldots\ (H_l\ w_1\ \ldots\ w_k))\rangle\}\},$$

   assuming $H_1, \ldots, H_l$ are new variables of appropriate types.

Notice that these substitutions are determined entirely by the heads of the flexible and rigid terms in question and they are finite in number.

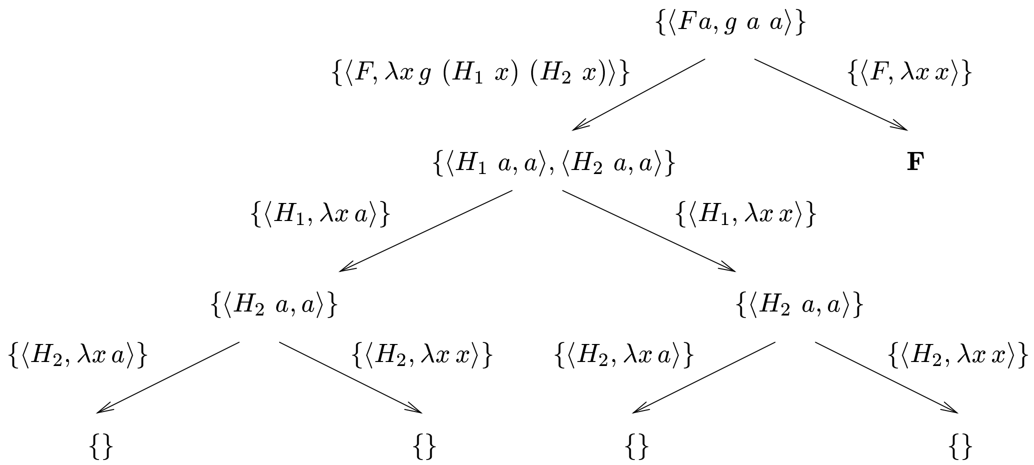

Figure 1: A matching tree for $\{\langle (F\ a), (g\ a\ a)\rangle\}$

The iterative use of the two described phases in unification naturally involves a search whose structure can be visualized through what has been called a matching tree [20]. Figure 1 presents such a tree for the unification problem $\{\langle F\ a, g\ a\ a\rangle\}$ encountered in Example 2.1. The arcs in this tree are labelled with the relevant imitation and projection substitutions and the nodes represent the result of transforming the set on the prior node by first applying the substitution on the incoming arc and then carrying out the simplification embodied in *SIMPL*. The leaves of a matching tree are labelled either with **F** or with a set of flexible-flexible pairs. A solution to the original unification problem can be obtained by composing the substitutions on the path to the latter kind of leaf with a unifier for that leaf. In the example presented, observing that an empty disagreement set has the empty substitution as its most general unifier, these solutions involve substituting $\lambda x\, g\ a\ a$, $\lambda x\, g\ a\ x$, $\lambda x\, g\ x\ a$ and $\lambda x\, g\ x\ x$ for F. A matching tree is exhaustive in the sense that the unifiers of the leaves of a completely expanded tree can be used in this fashion to produce all the unifiers of the original set. We note that, in particular instances, such a tree may include nonterminating branches and may, as a result, be infinite.

Our eventual objective is to describe a mechanism for solving goal formulas relative to given programs. The logical structure of a goal formula already engenders a search as we have seen in Section 2. In a refinement to the model presented there, unification problems arise in the attempt to solve atomic goal formulas. A state in the overall search may thus be described as a composite of a set of goal formulas and a disagreement set. Progress through this state space may be made by simplification steps applied to either the goal set or the disagreement set. In any given case, these steps must be relativized to a particular program $\mathcal{P}$. The notion of a $\mathcal{P}$-derivation [35] that generalizes SLD derivations described in [6] for first-order Horn clause logic makes this idea precise.

Definition 3.1. Let $\mathcal{P}$ be a program and let $\mathcal{G}$ and θ be symbols for sets of goal formulas and substitutions, respectively. Further, let $\mathcal{D}$ be a symbol for a disagreement set or the special value $\mathbf{F}$. Finally, let MATCH be a function on flexible-rigid disagreement pairs that produces the set of imitation and projection substitutions for any given pair. Then a tuple $\langle \mathcal{G}_2, \mathcal{D}_2, \theta_2 \rangle$ is said to be $\mathcal{P}$-derivable from a tuple $\langle \mathcal{G}_1, \mathcal{D}_1, \theta_1 \rangle$ in which $\mathcal{D}_1 \neq \mathbf{F}$ if it is obtainable from the latter by one of the following steps:

1. *Goal simplification step*: $\theta_2 = \emptyset$, $\mathcal{D}_2 = \mathcal{D}_1$, and for some $G \in \mathcal{G}_1$ it is the case that
   (a) G is $\top$ and $\mathcal{G}_2 = \mathcal{G}_1 - \{G\}$, or
   (b) G is $G_1 \wedge G_2$ and $\mathcal{G}_2 = (\mathcal{G}_1 - \{G\}) \cup \{G_1, G_2\}$, or
   (c) G is $G_1 \vee G_2$ and, for $i = 1$ or $i = 2$, $\mathcal{G}_2 = (\mathcal{G}_1 - \{G\}) \cup \{G_i\}$, or
   (d) G is $\Sigma\ P$ and $\mathcal{G}_2 = (\mathcal{G}_1 - \{G\}) \cup \{P\ Y\}$ where Y is a new variable.
2. *Backchaining step:* $\theta_2 = \emptyset$ and, for some rigid atom $G \in \mathcal{G}_1$ either
   (a) A is an atom obtained by instantiating the universal quantifiers in a clause in $\mathcal{P}$ with new variables and $\mathcal{G}_2 = \mathcal{G}_1 - \{G\}$ and $\mathcal{D}_2 = \mathit{SIMPL}(\mathcal{D}_1 \cup \{\langle G, A \rangle\})$, or
   (b) $G' \supset A$ is obtained by instantiating the universal quantifiers in a clause in $\mathcal{P}$ with new variables and $\mathcal{G}_2 = (\mathcal{G}_1 - \{G\}) \cup \{G'\}$, and $\mathcal{D}_2 = \mathit{SIMPL}(\mathcal{D}_1 \cup \{\langle G, A \rangle\})$.
3. *Flexible goal solution step:* $G \in \mathcal{G}_1$ is an atomic goal formula that has the (free) variable Y of type $\alpha_1 \to \cdots \to \alpha_n \to o$ as its head, and $\theta_2 = \{\langle Y, \lambda x_1 \ldots \lambda x_n\, \top \rangle\}$, $\mathcal{G}_2 = \theta_2(\mathcal{G}_1 - \{G\})$ and $\mathcal{D}_2 = \mathit{SIMPL}(\theta_2(\mathcal{D}_1))$; the application of a substitution to a goal set and a disagreement set here and below corresponds to its application to the component terms of these sets.
4. *Unification step:* For some flexible-rigid pair $\chi \in \mathcal{D}_1$, either $\mathit{MATCH}(\chi) = \emptyset$ and $\mathcal{D}_2 = \mathbf{F}$, or $\theta_2 \in \mathit{MATCH}(\chi)$ and $\mathcal{G}_2 = \theta_2(\mathcal{G}_1)$ and $\mathcal{D}_2 = \mathit{SIMPL}(\theta_2(\mathcal{D}_1))$.

A sequence of the form $\langle \mathcal{G}_i, \mathcal{D}_i, \theta_i \rangle_{1 \leq i \leq n}$ is a $\mathcal{P}$-derivation sequence for a goal formula G if $\mathcal{G}_1 = \{G\}$, $\mathcal{D}_1 = \emptyset$ and $\theta_1 = \emptyset$, and for $1 \leq j < n$, $\langle \mathcal{G}_{j+1}, \mathcal{D}_{j+1}, \theta_{j+1} \rangle$ is $\mathcal{P}$-derivable from $\langle \mathcal{G}_j, \mathcal{D}_j, \theta_j \rangle$. Such a sequence terminates in failure if $\mathcal{D}_n = \mathbf{F}$ and with success if $\mathcal{G}_n = \emptyset$ and $\mathcal{D}_n$ is either empty or contains only flexible-flexible pairs. In the latter case, we say that the sequence is a $\mathcal{P}$-derivation of G. Such a sequence embodies in it a solution to the query G in the context of the program $\mathcal{P}$ and the *answer substitution* corresponding to it is obtained by composing $\theta_n \circ \cdots \circ \theta_1$ with any unifier for $\mathcal{D}_n$ and restricting the resulting substitution to the free variables of G. □

An abstract interpreter for our language may be thought of as a procedure that, given a program $\mathcal{P}$, attempts to construct a $\mathcal{P}$-derivation for goal formulas. Such an interpreter would function by trying to extend an existing $\mathcal{P}$-derivation and will typically be faced with alternatives in this process. This interpreter can without loss of completeness choose to use a unification step whenever one is applicable. The only choices that are critical are, in fact, those of the disjunct to use when simplifying a disjunctive goal, the clause to use in a backchaining step, the substitution to use in a unification step and the point at which to solve a flexible goal. We assume a depth-first approach with the possibility of backtracking in the treatment of the first three aspects. The first two kinds of choices are present in a first-order language as well and similar methods can be used for treating them here; we note, in particular, that the choice of clauses in a backchaining step can narrowed down considerably by using simple techniques such as indexing on the names of predicates, a fact that we utilize in Section 6. The treatment of choices in the unification step and the bookkeeping mechanisms for realizing backtracking relative to these is a matter we discuss in a later section. Finally, we use a left to right processing order for goals that eventually determines the point at which flexible goals are encountered. This choice may, on occasion, lead to a loss of completeness but we believe this to be pragmatically justifiable.

4 The Representation of Lambda Terms

The abstract interpreter described in the last section embodies an idealization in its treatment of terms: it assumes a ready availability of head normal forms and an immediate access to the components of such forms. In reality, these normal forms must be computed. The efficiency of this computation and of the access to the structures of terms is mediated eventually by the representation chosen for these terms. We discuss the various factors influencing this choice below, motivating through this process the encoding that has been used in the *Teyjus* system. Our discussion also highlights the tradeoffs that are relevant to the representation question. Lambda terms evolve during computation in a manner that is difficult to predict statically, making experimentation with actual implementations a necessary component to quantifying the tradeoffs. An instrumented version of the *Teyjus* system is currently being used to obtain such an assessment. We indicate some of the observations from this study in our discussions here, leaving a detailed exposition to other papers.

4.1 The Representation of Bound Variables

Presentations of lambda terms usually employ a name-based rendition of bound variables. When such a representation is used also in an implementation, it becomes necessary to consider the α-conversion rule in comparison operations. For example, a common calculation within higher-order unification is determining whether the heads of two rigid terms are identical. Thus, suppose that we desire to unify the terms $\lambda y_1 \ldots \lambda y_n\, (y_i\ t_1\ \ldots\ t_m)$ and $\lambda z_1 \ldots \lambda z_n\, (z_i\ s_1\ \ldots\ s_m)$. Term simplification reduces this task to that of unifying the set

$$\{\langle \lambda y_1 \ldots \lambda y_n\, t_1, \lambda z_1 \ldots \lambda z_n\, s_1\rangle, \ldots, \langle \lambda y_1 \ldots \lambda y_n\, t_m, \lambda z_1 \ldots \lambda z_n\, s_m\rangle\}.$$

However, a prelude to effecting this transformation is recognizing that the heads of the two terms match and this clearly involves a renaming operation under the chosen representation.

If the kind of comparison described above arises often in computation, it is desirable to use a representation for terms that eliminates the need for bound variable renaming.

A scheme that is suitable from this perspective has been devised by de Bruijn [9]. The essential idea underlying this scheme is to manifest the connection between binding and bound occurrences of variables in lambda terms not through names but by using indices at the bound occurrences that count the number of abstractions in a parse structure of the term up to the one binding the occurrence. As an example, the term $\lambda x\, ((\lambda y\, \lambda z\, y\ x)\ (\lambda w\, x))$ is denoted using the de Bruijn approach by the expression $\lambda\, ((\lambda\, \lambda\, \#2\ \#3)\ (\lambda\, \#2))$, where $\#i$ is the representation of index i. Subterms of a term may have bound variable occurrences that are free in the local context. An occurrence of this kind is indicated by an index whose value exceeds the number of abstractions it is embedded under. Thus, in the subterm $(\lambda\, \#2)$ of the term considered above, the index $\#2$ corresponds to a bound variable occurrence that is locally free. The original de Bruijn encoding describes also a translation of globally free variable occurrences to indices. This part of the scheme is, however, not useful in our context. Globally free variables correspond in our computational model to variables that can be instantiated. A characteristic of all the substitutions that we consider for such variables is that the only unbound variables they contain are ones that are once again globally free; this property holds, for instance, of the imitation and projection substitutions discussed in the last section. Given this, these variables are best treated, in the usual logic programming style, as pointers to cells in memory that are tagged as unbound variables with instantiations being realized immediately by changing the contents of these cells.

The above discussion indicates a difference in representation and treatment at a pragmatic level between two kinds of variables that are similar in the underlying logic. Terminology that distinguishes between these variables will also be convenient in exposition. We henceforth use the expression 'logic variable' for a variable that is globally free, *i.e.*, is not bound by any explicit abstraction, reserving the terms 'bound variable' and 'free variable' for those variables that may be bound or free in a local context but that are ultimately captured by an abstraction and hence represented by a de Bruijn index.

The de Bruijn representation solves the problem mentioned at the outset. Thus, the two terms considered at the beginning are translated under this scheme into the de Bruijn terms $\lambda \ldots \lambda\, (\#i\ \hat{t}_1\ \ldots\ \hat{t}_m)$ and $\lambda \ldots \lambda\, (\#i\ \hat{s}_1\ \ldots\ \hat{s}_m)$, where, for $1 \leq i \leq m$, $\hat{t}_i$ and $\hat{s}_i$ are the de Bruijn representations of t_i and s_i respectively. Note that the heads of the two terms are identical under this representation. In general, the check for compatibility of the heads of two rigid terms that must be performed in the term simplification phase of unification becomes an identity test and α-conversion need never be considered.

The de Bruijn notation has another significant benefit in that it allows the abstractions that appear at the front of terms to be dispensed with in several situations. Such abstractions are often used in the unification process to encode the contexts in which to view the two terms that are to be unified. When these contexts are identical, as would be the case under the de Bruijn scheme, they can be left implicit. To understand the pragmatic impact of this observation, consider again the task of unifying the terms $\lambda \ldots \lambda\, (\#i\ \hat{t}_1\ \ldots\ \hat{t}_m)$ and $\lambda \ldots \lambda\, (\#i\ \hat{s}_1\ \ldots\ \hat{s}_m)$. This task can be reduced simply to that of unifying the set $\{\langle \hat{t}_1, \hat{s}_1 \rangle, \ldots, \langle \hat{t}_m, \hat{s}_m \rangle\}$, *i.e.*, the outer abstractions do not need to be appended to the front of the argument terms. Term simplification thus takes a form that is closely related to the first-order version: if the heads of the two rigid terms being considered are identical, the problem simply becomes one of recursively unifying their arguments. In contrast to the situation where the outer abstractions need to be replicated and added in front of the arguments, this transformation is one that can be easily implemented in a low-level abstract machine.

Although the de Bruijn notation obviates α-conversion in the determination of equality, renaming or, more precisely, renumbering is still necessary in the correct realization of β-contraction. To understand this, consider the reduction of the term $\lambda x\, ((\lambda y\, \lambda z\, y\ x)\ (\lambda w\, x))$ whose de Bruijn representation, as we have seen, is $\lambda\, ((\lambda\, \lambda\, \#2\ \#3)\ (\lambda\, \#2))$. This term reduces to $\lambda x\, \lambda z\, ((\lambda w\, x)\ x)$, a term whose de Bruijn representation is $\lambda\, \lambda\, ((\lambda\, \#3)\ \#2)$. Analyzing the reduction process in more detail, we observe the following. First, β-contraction may lead to a term with free variables being substituted inside abstraction contexts and, in this case, the indices for these variables have to be renumbered to avoid inadvertent capture. In the particular example considered, the term $(\lambda\, \#2)$ needs to be substituted inside an abstraction and in the course of doing this the index 2 representing a locally free variable occurrence has to be incremented by 1 to preserve the original binding relationships. Further, β-contraction eliminates an abstraction and the indices for variable occurrences that were free in that context have to be decremented by 1 to account for the disappearance of an intervening abstraction. In the example considered, this is reflected in the renumbering of the index 3 in the body of the redex to 2. These observations underlie the characterization in [9] of a substitution operation and the subsequent formalization of β-contraction.

Renaming is, of course, also necessary in realizing β-contraction under a name based representation and it is useful to contrast the effort that must be expended in the two cases towards understanding the tradeoffs between the different treatments of bound variables more precisely. As we have just seen, the β-contraction related renumbering under the de Bruijn scheme affects both the body and the argument parts of a redex. However, the renumbering work relating to the body can be significantly reduced by performing it simultaneously with the substitution of the argument of the redex into the body. Now, renaming when a name based representation is used affects only the body of a redex and not its argument part. Thus, in β-contracting a term of the form $(\lambda x\, t_1)\ t_2$, it is necessary to consider renaming only the variables explicitly bound within the subterm t_1. Even this kind of renaming can be avoided if it can be determined that the names of these bound variables do not clash with those of the free variables in t_2. However determining this requires a traversal of the argument part of the redex to calculate the set of variables that are free in it. A more efficient approach, used, for instance, in [2], is to always rename but, in a manner similar to the de Bruijn case, to fold such renamings into the same structure traversal that realizes the β-contraction substitution.

From the above discussion, it appears that there is an overhead in implementing β-contraction under the de Bruijn scheme that is roughly proportional to the effort required for renumbering the argument parts of β-redexes. We believe that this effort is small in practice for two reasons. First, actual renumbering can be often be finessed. For example, if there are no externally bound variables in the argument of the β-redex or if substitution is not made into a context embedded under abstractions, then renumbering is actually vacuous. This, in fact, is often the situation under a popular style of programming in λProlog [26], and other features of the lambda term representation that we describe in this section allow such properties to be recognized and utilized in reduction. Second, not all the cases where a nontrivial renumbering needs to be done constitute an extra cost. In general, when a term is substituted in, it is necessary also to examine its structure and possibly reduce it to an appropriate normal form. The necessary renumbering can, in this case, be incorporated into the same walk as the one that carries out this introspection. The main drawback of this approach is that it leads to a loss of sharing in reduction if the same term is substituted, and reduced, in more than one place since the required renumbering may be different in each of these contexts. However, empirical evidence suggests that the actual loss of such sharing is negligible [23], indicating thereby that any renumbering can be profitably folded into a required reduction walk.

In summary, then, the de Bruijn representation of bound variables has little real drawback in realizing β-contraction and significant advantages in checking identity modulo α-conversion and implementing higher-order unification. It has been used for this reason in the *Teyjus* implementation and we orient the rest of our discussion of term representation around it.

4.2 Encoding Substitutions in Terms

The effecting of substitutions over terms plays a central role in realizing the β-contraction operation, both from the perspective of actually contracting β-redexes and from that of adjusting indices or renaming bound variables. A desirable feature in implementing such substitutions is the ability to perform them in a lazy fashion. Several benefits can be gained from this ability. For example, consider the task of determining whether the (de Bruijn) terms $((\lambda\,\lambda\,\lambda\,\#3\ \#2\ s)\ (\lambda\,\#1))$ and $((\lambda\,\lambda\,\lambda\,\#3\ \#1\ t)\ (\lambda\,\#1))$ can be unified. We assume that s and t denote arbitrary terms here. We can conclude that these two terms cannot be unified by observing that they reduce respectively to $\lambda\,\lambda\,\#2\ s'$ and $\lambda\,\lambda\,\#1\ t'$, where s' and t' are terms that result from s and t by appropriate substitutions. Note that in making this determination, it is not necessary to explicitly calculate the results of the substitutions over the terms s and t. However, to achieve this conservation of effort it is necessary to be able to represent s' and t' as combinations of s and t with relevant substitutions. Similarly, consider the reduction of a term of the form $((\lambda\,((\lambda\,t_1)\ t_2))\ t_3)$ to head-normal form. Let t_2' be the term obtained from t_2 by substituting t_3 for the first free variable and decrementing the indices of all the other free variables by one. Then producing the head-normal form involves substituting t_2' and t_3 for the first and second free variables in t_1 and decrementing the indices of all other free variables by two. Each of these substitutions essentially involves a walk over the *same* structure, *i.e.*, the structure of t_1. It would obviously be beneficial if all these traversals could be combined into one. The ability to combine walks in this manner depends, once again, on the possibility of temporarily suspending a substitution generated by a β-contraction.

The delaying of substitutions has, in fact, been used extensively in the implementation of functional programming languages (*e.g.*, see [11, 14, 19]). In these contexts, the necessary delaying is realized by the simple device of combining a term with an environment that represents bindings for free variables that occur in it. When the de Bruijn representation is used, this simple device is adequate only if the overall term is closed and if subterms embedded within abstractions need not be explored. These assumptions are acceptable in the implementation of functional programming languages but are, unfortunately, not applicable to the context of interest to us. For example, the production of the head-normal forms that are needed during unification may well require the contraction of β-redexes embedded within abstractions as well as the propagation of substitutions under abstractions. In these cases, a more complicated substitution operation needs to be encoded. Thus, suppose that we need to β-contract a term of the form $(\lambda\, t)\ s$ that appears embedded within some abstractions. Now, t might contain variables that are bound by outside abstractions. If the result of contracting this β-redex is to be encoded by the term t and an 'environment', the environment must record not just the substitution of s for the first free variable in t but also the decrementing of the indices corresponding to all the other free variables. Similarly, imagine that we wish to propagate an environment (or, alternatively, a substitution for free variables) under the abstraction in a term of the form $\lambda\, t$. If the result is to be represented by a term of the form $\lambda\, t'$ where t' is itself encoded as t and an environment, then this environment must be obtained from the earlier one by 'shifting up' the index for the variables to be substituted for by one and adding an identity substitution for the variable with index 1. Further, the indices of the free variables in the terms that appear in the environment must themselves be incremented by 1.

Explicit substitution notations that have been developed in recent years for the lambda calculus offer a complete treatment of this kind of encoding of substitutions [1, 7, 16, 21, 39]. We outline here a version of such a notation that we have developed for use specifically in the implementation of our higher-order language [31]. This notation builds on the traditional de Bruijn notation by adding a new category of terms called a suspension. A suspension represents a 'skeletal' term together with a suspended substitution. Such a term has the structure $[\![t, ol, nl, e]\!]$, where t is a term, ol and nl are natural numbers and e is an environment. This suspension corresponds, intuitively, to a term t that used to occur inside ol abstractions but that now appears within nl of them. In generating the underlying de Bruijn term, therefore, the bound variables with indices greater than ol have to have their index values adjusted by the difference between ol and nl. Substitutions for the first ol bound variables are, on the other hand, contained in the environment e. Conceptually, the elements of such an environment are either substitution terms generated by a contraction or are dummy substitutions corresponding to abstractions that persist in an outer context. However, some renumbering of indices may have to be done at the place of actual substitution. To encode this renumbering, each element of the environment is annotated with the number of remaining abstractions under which the abstraction relevant to that element appears. This relative 'embedding level' can be used together with the overall embedding level nl to completely determine the needed renumbering.

The syntax of lambda terms in the new notation is, in fact, given by the category $\langle T\rangle$

(β_s) $(\lambda t_1)\ t_2 \longrightarrow [\![t_1, 1, 0, (t_2, 0) :: nil]\!]$

(β_s') $(\lambda[\![t_1, ol+1, nl+1, @nl :: e]\!])\ t_2 \longrightarrow [\![t_1, ol+1, nl, (t_2, nl) :: e]\!]$

(r1) $[\![c, ol, nl, e]\!] \longrightarrow c$, provided c is a constant.

(r2) $[\![x, ol, nl, e]\!] \longrightarrow x$, provided x is a logic variable.

(r3) $[\![\#i, ol, nl, e]\!] \longrightarrow \#j$, provided $i > ol$ and $j = i - ol + nl$.

(r4) $[\![\#i, ol, nl, e]\!] \longrightarrow \#j$, provided $i \leq ol$ and $e[i] = @(l)$ and $j = nl - l$.

(r5) $[\![\#i, ol, nl, e]\!] \longrightarrow [\![t, 0, nl - l, nil]\!]$,
provided $i \leq ol$ and $e[i] = (t, l)$ and $j = nl - l$.

(r6) $[\![t_1\ t_2, ol, nl, e]\!] \longrightarrow [\![t_1, ol, nl, e]\!]\ [\![t_2, ol, nl, e]\!]$.

(r7) $[\![\lambda t, ol, nl, e]\!] \longrightarrow \lambda[\![t, ol+1, nl+1, @nl :: e]\!]$.

(r8) $[\![[\![t, ol, nl, e]\!], 0, nl', nil]\!] \longrightarrow [\![t, ol, nl + nl', e]\!]$.

(r9) $[\![t, 0, 0, nil]\!] \longrightarrow t$.

Figure 2: Rule schemata for rewriting terms in the suspension notation

defined by the following rules:

$$\begin{array}{lcl}\langle ET\rangle & ::= & @\langle N\rangle \mid (\langle T\rangle, \langle N\rangle) \\ \langle E\rangle & ::= & nil \mid \langle ET\rangle :: \langle E\rangle \\ \langle T\rangle & ::= & \langle C\rangle \mid \langle V\rangle \mid \#\langle I\rangle \mid (\langle T\rangle\ \langle T\rangle) \mid (\lambda\langle T\rangle) \mid [\![\langle T\rangle, \langle N\rangle, \langle N\rangle, \langle E\rangle]\!]\end{array}$$

In these rules, $\langle C\rangle$ and $\langle V\rangle$ represent constants and logic variables, $\langle I\rangle$ is the category of positive numbers and $\langle N\rangle$ is the category of natural numbers. Further, $\langle E\rangle$ and $\langle ET\rangle$ are to be read as the categories of environments and environment terms, respectively. Terms of the form $[\![t, i, j, e]\!]$ must satisfy certain wellformedness constraints that have a natural basis in our informal understanding of their content: viewing the environment e as a list, its length must be equal to i, each element of it of the form $@(l)$ must be such that $l < j$ and each element of the form (t, l) must be such that $l \leq j$.

In addition to the syntactic expressions, the suspension notation includes a collection of rewrite rule schemata whose purpose is to simulate β-contractions. These schemata are presented in Figure 2. In these rules we use the notation $e[i]$ to denote the i^{th} element of the environment. Of the rules presented, the ones labelled (β_s) and (β_s') generate the substitutions corresponding to the β-contraction rule on de Bruijn terms and the rules (r1)-(r9), referred to as the *reading rules*, serve to actually carry out these substitutions.

The (β_s') schema has a special place in the calculus: it makes possible the combination of substitutions arising from different β-contractions. To understand its use, let us consider the head normalization of the term $(\lambda\,((\lambda\,t_1)\ t_2))\ t_3$. As the first step in this process, we might produce the term $[\![(\lambda\,t_1)\ t_2, 1, 0, (t_3, 0) :: nil]\!]$. The substitution may now be percolated inwards using the reading rules so as to reveal a β-redex at the top level. This produces the term

$$(\lambda\, [\![t_1, 2, 1, @(0) :: (t_3, 0) :: nil]\!])\ [\![t_2, 1, 0, (t_3, 0) :: nil)]\!].$$

At this point the β_s' rule schema is applicable and using it produces the term

$$[\![t_1, 2, 0, ([\![t_2, 1, 0, (t_3, 0) :: nil]\!], 0) :: (t_3, 0) :: nil]\!].$$

Notice that the substitutions generated by contracting the two β-redexes have been combined at this point into one environment and can be performed in a single walk over the structure of t_1.

In the translation of a suspension, it will eventually be necessary to substitute the arguments of β-redexes for bound variable indices. This operation is carried out in our calculus by instances of the rule schema (r5). There is, in general, a necessity to renumber indices in the term being substituted in and this is manifest in the schema (r5) in the construction of a suitable suspension. The rule schemata (r8) and (r9) recognize special circumstances relative such renumbering. The schema (r9) allows vacuous renumbering to be eliminated. By so doing, this rule facilitates a continued sharing relative to the substituted term. The schema (r8), on the other hand, permits a nontrivial renumbering walk to be combined with a walk affecting substitutions arising out of earlier β-contractions. Uses of the schemata (r8) and (r9) can be folded into the application of the schema (r5) and this is actually done in the *Teyjus* implementation. An interesting aspect of our overall system is that by utilizing the (β_s'), (r8) and (r9) schemata within the control strategy for generating head-normal forms that we describe later, it is possible to eliminate almost all occurrences of nested suspensions in practice.[4] This has obvious consequences with respect to the sharing of substitution walks.

While there is a case in principle for the laziness in performing substitutions, it is still necessary to determine how this plays out in practice. In situations where lambda terms are used in an essential way in programming, empirical studies indicate that using the suspension notation and the rules (β_s') and (r8) judiciously can reduce substitution walks to between a third and an eighth of what is needed when substitutions are performed eagerly [23]. There is a noticeable reduction in computation time as a result, up to 35%—measured over all computations, including backchaining over logic programming clauses—in some important cases.

4.3 A Dependence Annotation on Terms

There is a refinement to the suspension notation that can have practical benefits. This refinement consists of annotating terms to determine whether or not they contain variables bound by external abstractions. Referring to the two categories of terms as open and closed with obvious connotations, these annotations can be determined statically for de Bruijn

[4] The only ones that remain are, in fact, those that arise from our treatment of the η-conversion rule discussed in Section 5.1.

terms as follows. At the atomic level, de Bruijn indices are open whereas logic variables and constants are closed. For complex terms, an application is open if either its 'function' or 'argument' part is open and is closed otherwise, and an abstraction is open exactly when there is a bound variable occurrence within its scope that has a (relative) index greater than 1. Rewrite rules that transform terms in the course of computation can be modified in a straightforward way to maintain and propagate these annotations. For example, if a β-redex of the form $(\lambda\, t_1)\ t_2$ is closed, then the suspension $[\![t_1, 1, 0, (t_2, 0) :: nil]\!]$ that is generated from it must also be closed. Similarly, given a suspension of the form $[\![t_1\ t_2, ol, nl, e]\!]$ that is closed, the two top-level components of the term $([\![t_1, ol, nl, e]\!]\ [\![t_2, ol, nl, e]\!])$ that is obtained from distributing the substitution over the application must be closed. A complete presentation of these refined rewrite rules and a characterization of their properties may be found in [31].

The cost of maintaining the annotations discussed can be made small by using suitable low-level devices. In the emulator that is part of the *Teyjus* system, for example, an otherwise unused low-end bit is employed to indicate the annotation and the determination and setting of its value is folded into the overall manipulation of term tags. The advantage of maintaining annotations is at least twofold. First, the rewriting effort in determining the head normal form of a given term can be reduced. For example, consider a term of the form $[\![t, i, j, e]\!]$ where it is known that t is a term that is not dependent on outside abstractions. Then this term can be simplified immediately to (a pointer to) t. Second, this kind of simplification can foster a greater sharing of terms and, consequently, of rewriting steps. Thus, consider, once again, the term $[\![t, ol, nl, e]\!]$, but this time assuming that t is of the form $(t_1\ t_2)$ that may possibly be shared with other contexts. Attempting to reduce this term to head normal form in a situation where annotations are not used would result in the production of the term $([\![t_1, ol, nl, e]\!]\ [\![t_2, ol, nl, e]\!])$, in the process breaking the sharing over t. In contrast, with the use of annotations, the given suspension term will be simplified immediately to t and the subsequent reduction of t will be shared with all the other contexts in which it is used.

An obvious question is if the virtues of annotations are relevant to realistic computations in our higher-order language. We mention two situations in which these could be of benefit. In the first instance, observe that the substitutions that are computed by the higher-order unification process are actually closed in the sense discussed. Now, if occurrences of the variables being substituted for appear embedded within β-redexes, then the propagation of reduction substitutions over instantiations of these variables can be calculated trivially by utilizing annotations. As another example consider a β-redex of the form $(\lambda\, t_1)\ t_2$ where t_2 is a closed term as would be the case if this term appears statically at the top-level. The contraction of this term yields the suspension $[\![t_1, 1, 0, (t_2, 0) :: nil]\!]$. The percolation of the substitution of t_2 over the structure of t_1 might eventually lead to the replacement of a bound variable index by t_2. With reference to the rules in Figure 2, this replacement would produce a term of the form $[\![t_2, 0, l, nil]\!]$, *i.e.*, a term that corresponds to t_2 with a suitable renumbering of indices corresponding to free variable occurrences. By utilizing the fact that t_2 is known to be closed, the renumbering can be effected trivially. Furthermore, the bound variable that t_2 needs to be substituted for may occur in more than one place within t_1. In this case the use of the annotation on t_2 will also be responsible for the preservation of a meaningful sharing opportunity.

At an empirical level, we have observed that the use of annotations yields a substantial speedup relative to an eager approach to propagating reduction substitutions, the reduction in computation time being over 70% in several cases. While there is still a payoff from annotations when laziness in substitution and the combination of substitution walks as described in Section 4.2 are used, these appear to be of a much smaller kind. Thus, certain optimizations seem to overlap with others and a precise understanding of these interactions is needed.

4.4 The Implementation of Reduction

An issue of obvious importance is the order in which various operations are to be carried out on terms. From the perspective of unification, the main requirement is that of transforming terms into a form in which the head is exposed; in particular, the arguments of terms may be left in the form of suspensions. The idea of head-normal forms has been generalized to the suspension notation and its relationship to the conventional understanding of this notion has been explored in [31] as a prelude to its use in unification. At an implementation level, a strategy that might be used is one that produces these head normal forms only on demand and that does this by repeatedly rewriting the leftmost, outermost redex relative to the rules in Figure 2 till such time that an atomic head is revealed, possibly embedded under some abstractions. This strategy is an obvious generalization of the one used for rewriting β-redexes towards producing head normal forms in the usual setting and also has practical advantages: it provides the basis for delaying substitution walks as discussed in Section 4.2 and the different possibilities for combining term traversals during substitution and the adjustment of indices present themselves within it as well-defined choices between rewrite rules applicable at the same time. We discuss the realization of this approach within the *Teyjus* system further in Section 5.1.

Another issue to consider is whether to implement the rewriting of terms in a destructive or non-destructive manner. To understand the tradeoffs involved, let us consider the reduction of the term $\lambda\,(((\lambda\, t_3)\ t_2)\ t_1)$, in which t_1, t_2 and t_3 are arbitrary terms, to head normal form. Anticipating a discussion of internal representations, we may depict terms by graphical structures. Each term in such a representation translates to a node labelled with its category and containing its fixed length parts and pointers to relevant subterms and environments. Assuming such a visualization, the internal structure of the term of interest may be shown by the graph in the left half of Figure 3. Now, this term has a β-redex, given by the subterm $(\lambda\, t_3)\ t_2$, that has to be β-contracted in producing a head-normal form. If a destructive implementation of reduction is used, then this rewriting step will be effected by replacing the β-redex in place by the term $[\![t_3, 1, 0, (t_2, 0) :: nil]\!]$.[5] The consequences of this

[5]The representation of different kinds of terms generally require different amounts of space in an concrete realization. In this case, a destructive change may be achieved by using a special kind of term that serves

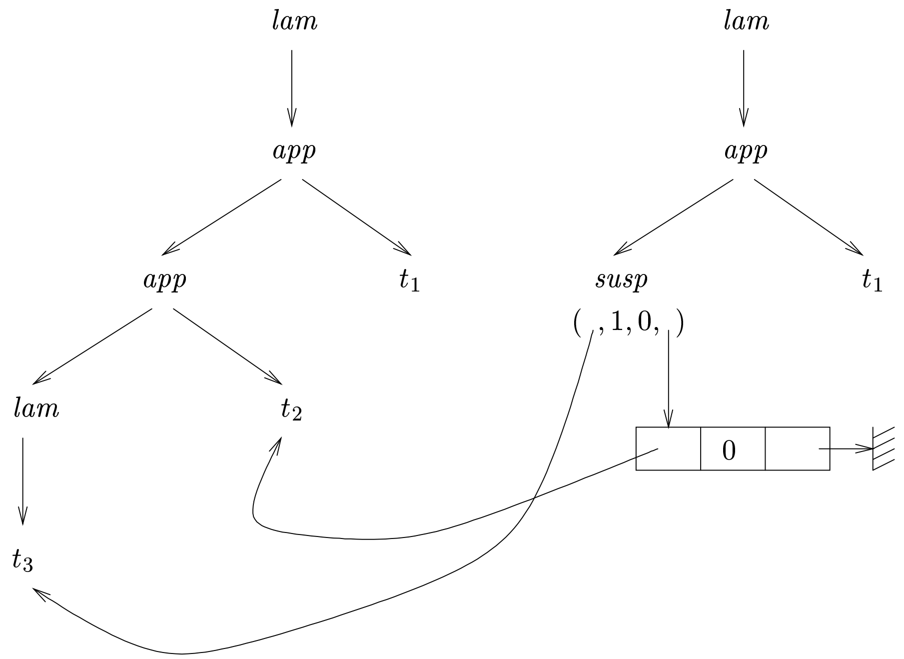

Figure 3: Non-destructive reduction of lambda terms

replacement will be felt immediately in all places where the β-redex appears as a subterm. If the β-contraction is done non-destructively, on the other hand, the subterm would be left intact and a new subterm of the form $[\![t_3, 1, 0, (t_2, 0) :: nil)]\!]$ would be produced. To obtain the effect of this rewriting step in the overall context, it would now be necessary to copy the structure of the term within which the β-redex is embedded. Thus, a non-destructive implementation of the β-contraction operation would eventually have to produce the structure shown in the right half of Figure 3.

Based on the above understanding, a destructive implementation appears to be an obvious choice in a context where such reduction is deterministic, *i.e.*, where future events may not require rewriting steps to be undone. The in-place replacement obviates a copying of the embedding context, a computation that obviously consumes time and space. Furthermore, it is only such a replacement that admits of any possibility of sharing in reduction; the effect of replacing $(\lambda\, t_3)\ t_2$ with $[\![t_3, 1, 0, (t_2, 0) :: nil]\!]$ will be felt in other contexts only if the term is changed physically at the place where these point to. A non-destructive implementation actually has an additional cost that is inhibiting. Consider, for instance, an attempt to head-normalize a term that is already in head-normal form; such a computation may be needed if the intent is to access the head normal form of an arbitrary term, a situation that arises naturally in our logic programming setting. Since copying of structure is needed in

as a 'reference': thus, the application cell corresponding to the β-redex may be changed into a reference to a newly created suspension term.

some cases, a naive implementation might simply replicate the structure of the term even when its subparts are unchanged. However, this is undesirable: a mere 'look-up' should not cause a new structure to be created. This kind of a copying can be avoided by putting explicit checks into the normalization procedure to determine when copying is necessary. This incurs a time penalty that does not arise in a destructive implementation.

Our interest is, of course, eventually in a context where backtracking over reduction computations may be necessary. As a specific example, a β-redex may manifest itself in a term as a result of a substitution for a variable that may have to be repealed at a later point. In this situation, a destructive implementation has the drawback that the in-place changes may have to be trailed to facilitate a subsequent resetting of state. A interesting point to note is that the form of trailing that is needed here is one that saves old values of cells and not simply the pointers to affected cells as in conventional Prolog implementations. The efficiency of such an implementation can obviously be improved by mechanisms that detect redundancy in trailing. The simplest method that can be used for this purpose is one that compares the location in heap of the term being changed with the most recent heap backtrack point to decide the necessity for trailing. Controlled forms of eagerness that push necessary rewriting steps to before the setting up of choice points are also useful. Another aspect that bears careful investigation is the possibility of committing to heap a cascade of reduction steps, such as those corresponding to a β-contraction and the propagation of the substitution it generates, only at the very end, thereby obviating the retraction of intermediate steps. The present version of the *Teyjus* system employs a graph based, destructive implementation of reduction and, as such, provides an effective vehicle for experimenting with these various possibilities.

The implementation of logic programming languages have, in the past, consider two different methods for the treatment of terms appearing in program clauses. In the structure sharing approach, these are represented as a combination of a fixed structure and bindings for variables [44]. In the more popular structure copying approach, entirely new copies of these terms are created through compiled code in each instance of use [43]. The destructive implementation of reduction is compatible only with this structure copying approach and we therefore assume its use in later sections.

4.5 Internal Representation

The final issue that we consider is the low-level representation of terms that embodies the various mechanisms that we have described. The most natural such encoding is one that uses a cell bearing a tag that indicates the relevant syntactic category for each term and that is complemented by additional cells containing information about further components. In the case that the dependency annotations discussed in Section 4.3 are used, it is best if the information they provide can also be accessed independently of the term category. In the emulator underlying the *Teyjus* implementation, this purpose is achieved by reserving the low-end bit of the tag bearing cell for these annotations.

The information that needs to be provided in addition to the term category depends, of course, on the category of the term. If this is a constant or a bound variable, all that is needed is a pointer to the descriptor for the constant or the index, and this can be folded into the cell bearing the tag. In the case of a logic variable that is as yet uninstantiated, the contents of the cell are unimportant and the instantiation of such a variable is realized by changing the contents of this cell to correspond to one of the other term categories. An abstraction cell must contain a pointer to the body of the abstraction. A suspension term requires the maintenance of its two indices, a pointer to the skeletal term and a pointer to its environment. An environment can be represented as a list and, in this form, admits of considerable sharing. For example, consider the suspension

$$[\![(t_1, 2, 0, ([\![t_2, 1, 0, (t_3, 0) :: nil]\!], 0) :: (t_3, 0) :: nil)]\!]$$

that arises in the course of normalizing the term $(\lambda\,(\lambda\,t_1)\;t_2)\;t_3$. The final part of the environment of the overall term given by $(t_3, 0) :: nil$ and the environment of the term $[\![t_2, 1, 0, (t_3, 0) :: nil]\!]$ arise from the same source and can therefore be represented as pointers to the same structure.

The representation of the final category of terms, namely, applications, requires more care. The most natural, and perhaps the conceptually clearest, approach is to utilize the curried structure, rendering each application into a pair of pointers to its function and argument parts. Unfortunately, this kind of rendition incurs a high cost in the most common form of access to terms. The objective with terms is typically to get to the heads of their head normal forms. Further, operations such as term simplification in unification are best realized if the arguments in a head normal form are available as a vector. Suppose that we have a term that at compile time has the structure $\lambda\,\ldots\,\lambda\,(h\;t_1\ldots t_n)$. If a curried representation is used for this term, n applications will have to be traversed before the head is reached and a vector of arguments will also have to be explicitly constructed in the course of this descent under application structure.

An alternative encoding of application that is reminiscent of the treatment of terms in conventional logic programming implementations is to translate it into a structure containing three components: a function part, a pointer to a *vector* of arguments and an 'arity' that indicates the size of the arguments vector. Such a representation has especially nice properties when the program at hand is a first order one. In this case the top-level structure of every compound (application) term that is encountered during computation is already available at compile time. Thus, the head normal forms of these terms are available without any reduction calculations and the described representation allows a quick determination of this fact as well as an immediate access to the functor and arguments parts. These appear to be important properties since efficiency over first-order like computations is significant even to a higher-order logic programming language [25].

Our low-level representation for terms comes close to the one generally employed for first-order terms with the described encoding of applications. However, there are still differences that should be mentioned. One difference concerns the specific structure chosen for compound terms. In the first-order case, internal nodes in the tree representation of terms cannot change. This fact can be exploited to fold the functor and arguments parts into one vector and, thereby, to reduce compound terms to a single pointer. A similar optimization is not possible in the higher-order context. The contraction of a β-redex, for instance, transforms an n-fold application into an $(n-1)$-fold one and there must be sufficient flexibility in the encoding of terms to capture this situation. The other difference lies in the registration of destructive changes for the purposes of backtracking. First-order terms evolve during computation only by virtue of bindings for logic variables. By picking a uniform representation for such variables, the state prior to such a change can be recorded simply by retaining a pointer to the cell for the corresponding variable. This kind of optimization is not available in the higher-order context when reduction is implemented destructively and, as we have already noted, the original value of the modified cell needs also to be remembered in order to resurrect the previous state.

Up to this point we have not considered explicitly the fact that the terms of interest to us have types associated with them. These types have a twofold role in the language [38]. At one level, they serve to limit the set of acceptable programs. At another level, they participate in the computational mechanism of the language; this role is apparent from the manner in which types determine the imitation and projection substitutions that are to be generated for a flexible-rigid pair. The first function of types is relevant to compilation but does not affect the execution of a program and so does not have a bearing on runtime representations. As for the second purpose, we observe that it is sufficient to maintain types with only the constants and logic variables appearing in lambda terms. Maintaining such annotations is also necessary: the types of logic variables are needed for both the imitation and projection substitutions and the types of constants are needed in determining the imitation substitutions.

The need to maintain types adds a extra component—a pointer to a type—to the representation of logic variables. The representation of constants is unchanged since the type information can be combined with the other data comprising their descriptors. While we do not discuss this issue in detail here, in the presence of polymorphism, types are best represented by pointers to a type 'skeleton' and a type environment [22]. The treatment of polymorphism thus adds an extra cell to the representations of constants and logic variables.

5 Runtime Support for Higher-Order Unification

We now consider the task of supporting the enhanced notion of unification present in our language. The problems that have to be dealt with are threefold. First, it is necessary to consider the normalization of terms during execution. Second, states in our abstract interpreter are given also by disagreement sets and an efficient method for maintaining such sets explicitly is needed. Finally, higher-order unification has a branching character, a facet we realize through depth-first search with backtracking. In implementing this approach, it is necessary to identify the important components of state that need to be remembered and also to describe suitable encodings for such information. We discuss these various issues below and we describe approaches to accounting for them within an actual implementation.

5.1 Normalization of Terms

The simplification operation and the postulation of substitutions within unification depend on terms being presented in (a generalized) head-normal form. Terms can arise during computation that are not in this form; consider, for instance, the structure of a term after a substitution dictated by imitation or projection has been made for a variable of function type that appears in it. Mechanisms for normalizing terms are therefore needed as also is a protocol for deploying these at points where head normal forms are desired. As discussed in Section 4.4, a strategy that rewrites the leftmost, outermost redex at each stage is a natural one to use for head-normalization. The suspension notation allows the substitution generated by a β-contraction to be treated as a truly atomic operation and thereby facilitates an iterative, stack based realization of this strategy. Such an approach is embedded in the implementation of normalization within the *Teyjus* system. We sketch this component of the system below as a prelude to explaining its use in the overall computation scheme. A more detailed description of the reduction procedure may be found in [30].

The *Teyjus* implementation of head-normalization actually uses two stacks called the structures list or *SL* stack and the applications stack, the latter facilitating a destructive realization of reduction. Both stacks store references to terms and can share a common space in an abstract machine, with their tops growing towards each other. The reduction procedure looks at the term pointed to by the top of the *SL* stack and the value in a global register *NUMARGS* to determine its next step. At the outset, a reference to the term to be reduced is placed on the top of the *SL* stack and the *NUMARGS* register is set to 0. The main actions of the procedure are dependent on the term referenced by the top of the *SL* stack having a non-suspension structure. For this reason, if this term is a suspension, the first task becomes that of exposing such a form for it. This objective is achieved immediately using one of the reading rules in Figure 2 when the skeleton is itself not a suspension. Otherwise a non-suspension form must be exposed first for the skeleton and a simple iterative process that begins by placing a reference to this skeleton on the top of the *SL* stack serves to realize this. Eventually, when a non-suspension form is exposed, if the top of the *SL* stack is a reference to an application, this reference is recorded in the applications stack and is replaced in the *SL* stack by a sequence of references to its "operand" and "operator" parts, with the *NUMARGS* register being incremented by the number of operands. If the top of the *SL* stack contains a reference to an abstraction, the action taken depends on the value in the *NUMARGS* register. If this is 0, the *SL* stack reference is replaced by one to the body of the abstraction. Otherwise, a leftmost, outermost β-redex has been found and needs to be contracted. This action is realized by popping the top two items on the *SL* stack, using them to construct a suitable suspension a reference to which is pushed onto the *SL* stack, destructively updating the application available from the top of the applications stack and, finally, decrementing the *NUMARGS* register by 1 to account for the disappearance of an argument. The final possibility for the top of the *SL* stack is that it is a reference to an atomic term, *i.e.*, one that is a constant or a bound or free variable. This situation signals that a head normal form has been found and hence terminates the overall process.

To understand the integration of the head-normalization procedure into the larger computational framework, suppose that it is invoked with a term that can be reduced to the form $\lambda x_1 \ldots \lambda x_n\,(h\ t_1\ \ldots\ t_m)$, where h is a constant or variable. When the procedure is finished, the *SL* stack will contain, in consecutive locations from the top, references to the de Bruijn representation of h and the suspension representations of terms $s_1, \ldots, s_m$ that are β-convertible to $t_1, \ldots, t_m$. This kind of access to the body of the term is particularly convenient for the other operations required within unification. First of all, the heads of terms and their status, whether rigid or flexible, is easily determined. Further, assume that the simplification operation is to be applied to two terms t and r whose head-normal forms have identical binders. The head-normalization procedure can, in this case, be invoked to lay out the bodies of these two terms in different segments of the *SL* stack. Then, if the two terms are rigid and have identical heads, the terms out of which new disagreement pairs have to be formed appear at the same displacement from different starting locations, thereby facilitating an iterative structure to further processing. The availability of the arguments of the head-normalized term in contiguous locations turns out also to be important to the compilation model that we discuss in Section 6.

Our description of the term simplification process above assumes that the lengths of binders of two terms to be unified are identical. This situation may actually not hold automatically but, rather, may have to be achieved at the required points in computation by using the η-conversion rule. A few simple changes to our normalization routine suffices for making the requisite adjustments. Thus, suppose that we are comparing two terms that have as head-normal forms $\lambda x_1 \ldots \lambda x_n\,(c\ t_1\ \ldots\ t_l)$ and $\lambda x_1 \ldots \lambda x_m\,(c'\ s_1\ \ldots\ s_k)$. Our procedure can easily record the values of m and n when producing these forms. Now suppose that n is greater than m and that c' is identical to one of $x_1, \ldots, x_m$. Then the effect of adjusting the binder length of the second term to n on its head in a suspension representation of the terms can be captured simply by adding $n - m$ to the index value corresponding to c'. The changes to the first k arguments of this term under such an adjustment are also straightforward to capture: if s'_i is the term in suspension notation corresponding to s_i, the desired adjustment is encapsulated in the term $[\![s'_i, 0, j, nil]\!]$ where $j = n - k$. Finally, new arguments need to be added to the term, but this is particularly easy to do on-the-fly, they being just the de Bruijn indices $\#(n-m), \ldots, \#1$.

The normalization process requires the creation of new structures for terms at various points in its execution. These terms are best allocated on the heap in a WAM-like model and this is, in fact, what is done within the *Teyjus* implementation.

5.2 Explicit Representation of Disagreement Sets

Disagreement sets arise in principle even in the context of first-order unification. However, typical implementations of this operation avoid the explicit treatment of such sets by utilizing a recursive, depth-first processing of the subparts of the two terms that are to be unified. Careful attention to the order in which subterms are processed is known to make a substantial difference to the worst case behaviour [24], but the 'pathological' cases seldom seem to arise in practice. Given this, flexibility in choosing the next pair of (sub)terms is usually sacrificed for a simpler processing structure.

Two properties of first-order unification are actually critical to adopting the approach that treats disagreement sets implicitly: its decidability and the existence of most general unifiers. Neither of these properties carry over to the higher-order context. While it is still possible to use a recursive process that explores unifiers for subterms in a depth-first fashion, basing the unification computation entirely on such an approach appears pragmatically undesirable. In particular, in a situation where choices have to be made in substitutions, it appears best to bring all available constraints to bear on making them. Thus, suppose that it is necessary to unify two terms of the form $(f\ t_1\ \ldots\ t_n)$ and $(f\ s_1\ \ldots\ s_n)$, where f is a constant (function) symbol and, for $1 \leq i \leq n$, t_i and s_i are arbitrary terms. We may well attempt to do this by unifying the pairs $\langle t_1, s_1\rangle, \ldots, \langle t_n, s_n\rangle$ in sequence. Now, in the course of unifying the pair $\langle t_1, s_1\rangle$, it may be necessary to pick one of several substitutions for a variable x. This variable may appear in other pairs of subterms as well and several of the substitution choices for x may render these pairs non-unifiable. Using this information, at the very least, curtails the branching in search. In particular cases, this may even make a difference between finding and not finding a unifier: some choices of substitution for x that are ruled out by their effects on other pairs may lead to a neverending search when the pair $\langle t_1, s_1\rangle$ is considered in isolation.

A better approach to finding unifiers for a disagreement set, then, appears to be the following. At any point in the computation, we select a pair from the set and proceed to search for a unifier for only this pair till such time that more than one possibility needs to be considered in furthering the search. At such a point, we pick a possible substitution and examine its effect on the rest of the disagreement set before proceeding further. Implementing this approach clearly requires an explicit representation and manipulation of disagreement sets within the unification process. Actually, when the task of solving a goal in our higher-order language is considered in its entirety, it becomes clear that disagreement sets may at times have to be carried even across invocations to the unification procedure. Repeated applications of the unification step described in Section 3 may reduce a disagreement set to a form in which, while being nonempty, it contains only flexible-flexible pairs. As already noted, it is best to 'suspend' processing on such a set, reactivating it only after the application of backchaining steps has further constrained the unification problem.

We illustrate the preceding remarks relative to the *mapfun* predicate from Section 2 whose definition is reproduced below.

$$\begin{array}{l} \mathit{mapfun}\ \mathit{nil}\ F\ \mathit{nil}. \\ \mathit{mapfun}\ (X :: L1)\ F\ ((F\ X) :: L2)\ \texttt{:-}\ \mathit{mapfun}\ L1\ F\ L2. \end{array}$$

Suppose now that we are interested in solving the query

$$\mathit{mapfun}\ (a :: b :: \mathit{nil})\ G\ ((G\ b) :: (h\ a\ b) :: \mathit{nil})$$

The only applicable step within our abstract interpreter in this case is a backchaining one. The attempt to use the first clause quickly leads to a failure within the term simplification phase. Using the second clause, on the other hand, leads to the disagreement set

$$\{\langle X, a\rangle, \langle L1, b :: nil\rangle, \langle F, G\rangle, \langle (F\ X), (G\ b)\rangle, \langle L2, (h\ a\ b) :: nil\rangle\}.$$

Most of the pairs in this set can be 'solved' by simple substitutions. Thus, by substituting a for X, $b :: nil$ for $L1$, G for F, and $(h\ a\ b) :: nil$ for $L2$, this set can be reduced to the set $\{\langle (G\ a), (G\ b)\rangle\}$. The only remaining pair is a flexible-flexible one that cannot be further simplified[6] and so must be carried into the attempt to solve the subgoal

$$(mapfun\ (b :: nil)\ G\ ((h\ a\ b) :: nil)$$

corresponding to a suitably instantiated version of the body of the second clause. Now, this goal can be solved by using two different substitutions for G: $\lambda x\, (h\ a\ x)$ and $\lambda x\, (h\ a\ b)$. Instantiating G with either of these substitutions changes the status of the suspended disagreement pair which then turns out to be a 'filter' that rejects the first solution but accepts the second.

Assuming that disagreement sets must be represented explicitly at certain points, an important question to consider is that of the structure of their representation. The following considerations seem to be important in this context:

1. These sets evolve incrementally during computation. In particular, changes result from adding new pairs to an existing set or by effecting substitutions that modify only some pairs leaving the others unchanged. This suggests that a representation that allows a new disagreement set to reuse the unchanged portions of the set from which it arises, might work well in practice.

2. For efficiency in backtracking, it should be possible to rapidly recreate disagreement sets that were in existence earlier. This becomes a pertinent issue if the kind of sharing described in (1) is realized through destructive changes.

A representation for disagreement sets that suffices for meeting the above requirements is one based on doubly linked lists the elements of which are pairs of pointers to the terms constituting the pairs in the set. Given that these sets arise in the course of backchaining or clause invocation and finally disappear in the event of backtracking, the lists representing them are naturally allocated in the heap in a WAM-like setting. The need to examine disagreement sets during computation requires that the beginning of the lists representing them be recorded in machine states. A special register that we refer to as the live list or LL register and that contains a reference to the first element in the list at each execution point serves to realize this purpose. Now, there are two ways in which a disagreement set might change during computation. First, term simplification may require some element of the set to be removed and new pairs corresponding to subterms that need to be unified to be added to the set. The removal of a pair is realized in this setting by changing the 'after' and 'before' pointers of the elements on either side of it in the list representation. A

[6] There is actually a most general unifier for this pair: the substitution $\lambda x\, Y$ for G, where Y is a new variable. However, this substitution will not be considered by the abstract interpreter described in Section 3. More to the point, there is no systematic, non-redundant way to search for unifiers for flexible-flexible pairs.

subsequent re-inclusion of the removed pair into the set can be effected easily if a reference to it is maintained, something that can be done through a (properly annotated) entry in the trail stack. The addition of new pairs is also simple: entries for these can be created on the heap and added to the beginning of the live list. The second way in which a disagreement set may change is through a backtracking operation. To support this action, it is necessary also to store the contents of the LL register in choice points at the time of their creation. The relevant disagreement set can then be resurrected by utilizing information in the trail stack to restore deleted pairs and using the old value of the LL register to remove the pairs added beyond the point being backtracked to.

It is in principle possible to perform all the processing within the term simplification phase of unification using only the heap and the live list. However, judiciousness should be exercised in utilizing the heap since space allocated in it becomes unavailable until it is reclaimed through backtracking. With this in mind, we observe that when a rigid-rigid pair is encountered during term simplification, the processing can be applied recursively to the subterms and additions to the heap and the live list need not take place till a flexible-flexible or a flexible-rigid pair is encountered. Using this idea relative to the *mapfun* example considered earlier in this section, for instance, it necessary to contemplate an addition to the heap of only the pairs appearing in the set

$$\{\langle X, a\rangle, \langle L1, b :: \mathit{nil}\rangle, \langle F, G\rangle, \langle (F\ X), (G\ b)\rangle, \langle L2, (h\ a\ b) :: \mathit{nil}\rangle\},$$

and not of all the pairs that arise in the course of generating this set. When one of the terms in a disagreement pair is known statically, this kind of processing can be realized through special instructions and a compilation process similar to that used in the first order case, and we discuss this matter in greater detail in the next section. However, the two rigid terms in a pair can sometimes arise dynamically and term simplification has in this case to be carried out in 'interpretive' mode. In the context of a virtual machine based implementation, a special pushdown list in combination with an iterative code fragment can be used to realize this computation.

There are certain forms of disagreement pairs for which most general unifiers can be immediately identified. A simple example of this kind arises from first-order unification: given a pair of the form $\langle X, t\rangle$ where X is a variable of atomic type and t is a term in which X does not appear, all unifiers for the pair must be instances of the unifier that substitutes t for X. Alternatively, if X does occur in t, failure in unification can be registered immediately. This observation can be generalized to the higher-order context. Given a pair of the form $\langle \lambda x_1 \ldots \lambda x_n\, X, \lambda x_1 \ldots \lambda x_n\, t\rangle$, where X has an arbitrary type, X can be bound to t and this pair can be removed from the set provided neither X nor the variables in $\{x_1, \ldots x_n\}$ appear in t; interestingly, the verification of the proviso is simplified by the use of the de Bruijn notation. Note that the occurrence of X in t does not by itself signal failure in the higher-order case. However, the 'occurs-check' from the first order case can be generalized to a 'rigid path check' that detects the impossibility of unification in some cases and that simplifies the search for unifiers in other cases by binding X to a term that represents an initial 'section' of t and by adding pairs to the disagreement set to represent the remaining constraints on unifiers. Some flexible-flexible pairs can also be solved by this process. For example, the pair $\langle F, G\rangle$ that arose in the context of the *mapfun* example can be solved in this manner. Using observations such as these reduces the need to consider the general imitation and projection substitutions and hence also the attendant bookkeeping steps. In the case of the *mapfun* query, the disagreement set can in fact be reduced to $\{\langle (G\ a), (G\ b)\rangle\}$ by these means. Significantly, first-order unification can be solved immediately using these observations. Empirical studies indicate that a large number of the unification problems that arise even in the higher-order context fall into this category [25], suggesting the general importance of incorporating these observations into an implementation.[7]

The appropriate time to consider such substitutions is during the term simplification phase. Doing this and also being conservative in the additions to the heap now calls for the use of two pushdown lists. The general scheme works as follows. The simplification of a disagreement set proceeds as before with the use of the first pushdown list, except that the process may now also involve making bindings to variables. When the process 'bottoms out' with a flexible-flexible or flexible-rigid pair, this is pushed onto the top of the second pushdown list instead of the heap. When all the pairs in the original disagreement set have been simplified, it is checked whether any bindings were made in the course of simplification. If no bindings were made, the pairs in the second pushdown list are transferred to the heap and included in the live list. If, on the other hand, any bindings were made, the simplification process is repeated with the disagreement set being given now by the live list and the pairs in the second pushdown list and the roles of the two pushdown lists being reversed.

5.3 Recording Branch Points in Unification

A depth-first approach to exploring alternatives in unification requires that information be recorded at branch points that is sufficient for recreating state and for determining the remaining possibilities upon backtracking. The state information can actually be factored into two conceptual kinds: that which pertains to clause usage in the backchaining model of computation and that which relates to the unification problem, such as the disagreement set and the pair of flexible and rigid terms that are under consideration at a particular juncture. As we have already noted, the approach that we propose to use is one that solves unification problems to the extent possible before contemplating backchaining steps. Given this strategy, in situations where genuine higher-order unification is involved, a sequence of branch points are likely to be generated for which the state insofar as it pertains to clause usage is identical. Thus, if this information is represented separately from the rest of the backtracking data, it becomes possible to share it across more than one branch point.

In light of the above observations, we propose to record information relevant to branching in unification in two layers that we refer to, respectively, as the shared part and the variable

[7]The use of this first-order like processing in the higher-order case is dependent on variables being maintained in an 'η-reduced' form and care in the compilation process is therefore needed to preserve its applicability. Another interesting possibility is to extend this kind of processing to include a richer class of unification problems that also have a non-branching character and that have been observed to cover most useful programming applications of higher-order unification [26]. This matter is currently under investigation.

part. Assuming a WAM style compilation model, the most appropriate juncture at which to consider genuine higher-order unification is right after a compiled form of term simplification akin to first-order unification has been carried out relative to the head of the clause and before an attempt is made to solve the clause body. At this stage, if the LL register indicates a nonempty disagreement set, the first action would be to create the shared part of the unification branch point records that stores clause usage information. More specifically, this part would record at least the following state data:

- The program pointer that determines the instruction to be executed upon successful completion of this phase of unification.
- A pointer to the most recent environment record.
- The continuation pointer; this is relevant in the case of clauses with an atomic body for which an environment record does not exist containing this information.
- Argument registers that need to be preserved for use in the first goal in the clause body.

Additional information may have to be recorded in this part depending on auxiliary language features. For example, in a framework that permits the use of the *cut* control primitive, the contents of the cut point register that indicates the backtracking point up to which to eliminate choices needs to be stored as well. Similarly, it has been found useful to give the programmer dynamic control over whether projection or imitation substitutions are to be tried first within higher-order unification. In this situation, the regimen in effect at this instance should also be stored in the shared part for later restoration.

Once the shared part that has been constructed, a global reference to it is maintained in a special register that we refer to as the BRS register. Computation now proceeds to simplifying the disagreement set and, eventually, to picking a flexible-rigid pair whose imitation and projection substitutions have to be examined. After such a pair has been determined and a substitution for it has been selected, information must be left behind for examining the remaining alternatives in case of backtracking. This data is encoded in the variable part of unification branch point records that comprises the following components:

- The heads of the flexible and rigid terms together with their types.
- Information determining what substitutions remain to be tried. A simple way to encode this is by remembering the number of projection substitutions already tried; the additional knowledge of whether imitation or projection substitutions are being tried first completely determines the alternatives left.
- The contents of the LL register that will be used in consort with the information in the trail stack to restore the disagreement set on backtracking.[8]

[8]This component needs also to be added to the usual *choice point record* of the WAM that stores information for backtracking over clause choices.

- Pointers to the top of the heap and the trail stack that determine the status of these data areas.

- The contents of the *BRS* register for restoring clause context on backtracking.

- A pointer to a record of the preceding branch point in computation, to be used when all alternatives at this stage have been exhausted.

Although the heads of the flexible and rigid terms suffice for generating all the substitutions, certain operations have to be repeated on these in each case. Thus, the binder of every substitution that is constructed is identical. Similarly, the vector of arguments of the general arguments of the substitution terms are identical both within a single substitution and across the imitation and projections. Finally, the target type of the flexible head is used repeatedly in determining the appropriateness of each projection substitution. Assuming the acceptability of trading off space for time, these components may be computed once when the variable part is set up and references to them may be saved for later use. This is, in fact, the course adopted within the *Teyjus* implementation.

The successful selection of a substitution within the unification process is followed by another term simplification phase. If there is another flexible-rigid pair in the resulting disagreement set, further substitutions must be posited, leading to the setting up of the variable part of another unification branch point record. Note that the shared part of this record, pointed to by the *BRS* register, is the same as that for the previous such record. This process continues till eventually a failure is encountered or the disagreement set is reduced to a solved form. In the latter case, computation continues with an attempt to solve the next (predicate) goal through a backchaining process.

There is, of course, the possibility of failure along the path currently being explored. In this case backtracking must take place to the most recent choice point either in clause selection or in unification. In order to determine the appropriate such point, the records corresponding to them are chained into one linear sequence based on their age and a pointer to the most recent one is placed, as in the WAM, in the B or backtrack register. Now, certain actions, such as the unwinding of the trail stack, the resetting of the disagreement set and recovery of heap space, are identical regardless of whether computation returns to trying another clause or another unifier and can be carried out uniformly with a little coordination in the structures of the records corresponding to these different kinds of backtrack points. However, other actions, such as the generation of another substitution or the selection of another clause, do need a knowledge of the kind of choice being reconsidered. One approach to providing this information would be to mark each backtrack point record in a special way at the time of its creation. A more elegant solution is possible in a virtual machine and compilation based framework and is, in fact, used in *Teyjus*. In this system, a special instruction is included in the instruction set whose purpose is to utilize the information in the variable part of a unification branch point record to generate a new substitution, to reset the *BRS* register and the state reflecting clause usage context and to continue with the unification computation. The right backtracking action can now be achieved simply by storing a pointer to a program location containing this instruction in a field of the variable part of a unification branch point record that is coordinated with the next clause field of the usual choice point record of the WAM; a uniform transfer of control to the stored program point and the execution of the corresponding instruction then achieves the appropriate backtracking action.

Branching in computation is obviously costly both in time and in space and every effort should be expended to exploit deterministic execution patterns whenever possible. One approach to doing this within the unification computation is, as we have already mentioned, to build a treatment of more special cases in which most general unifiers exist into the term simplification process. Another useful idea is to employ quick dynamic tests to determine that no further substitutions exist in certain cases and to discard unification backtrack points eagerly on this basis. Some heuristics of this kind are embedded in the *Teyjus* system but this is a matter that deserves further attention.

A final point to mention concerns the allocation of space for the terms generated for projection and imitation substitutions. This is best done on the heap since backtracking permits the space to be reclaimed when the substitution itself becomes redundant.

6 An Abstract Machine and Compilation Model

The abstract machine for Prolog is designed to support a compiled treatment of the four main components of the underlying model of computation: the processing of the structure of complex, usually conjunctive, goals, the setting up of the arguments of atomic goals, the sequencing through clause choices for such goals and the unification of the arguments of these goals with the statically known arguments of a clause head. A further aspect that receives special attention is the detection of determinism. Nondeterminism is costly to deal with and the need to do so can often be eliminated by utilizing the structure of the actual arguments of atomic goals to prune choices early during execution. This observation is exploited in practice by including a special set of instructions that allow clause choices to be indexed by the arguments and by building the use of these instructions into the compilation process.

The basic issues in a first-order context persist also in our higher-order language. Much of the machinery and even the instruction set that are embedded in a WAM-like architecture for treating these aspects can, in fact, be carried over to the implementation task at hand. However, some new devices are needed, primarily for dealing with a richer structure for terms and a more complex unification operation. Moreover, there must be differences in the interpretation of some instructions. Instructions that examine the structure of terms must, for instance, have the ability to head-normalize these terms if this is needed during execution. Further, the instructions that realize unification completely in a first-order setting suffice only to implement the initial term simplification phase of higher-order unification. The capability to leave unification problems that cannot be solved in this manner to a later, interpretive phase should therefore be built into these instructions. Such a ‘deferring’ action should, of course, be complemented by an invocation of the remaining higher-order unification process at a suitably chosen point.

We present, in this section, an extended version of the WAM that develops on these ideas. We summarize first the modifications to the machine structure that were implicit in our discussion of the treatment of higher-order unification. We then describe changes to the instruction set. The last part of this section illustrates the compilation model by presenting the code generated for some simple higher-order programs. A familiarity with the original abstract machine of Warren, such as might be obtained from [3] or [43], is assumed in this exposition.

6.1 The Structure of the Extended Machine

Figure 4 depicts the various data areas and registers present in the extended abstract machine and provides a snapshot of a machine state during computation. The code area, the heap, the local stack and the trail of the WAM persist in this machine. The new data areas are the *SL* stack, the applications stack and the two pushdown lists. The first two components are utilized by the head-normalization code as described in Section 5. The pushdown lists are used in simplifying disagreement sets and help, as we have seen, in conserving heap space. We observe that only one of these pushdown lists is really new: one pushdown list is usually employed by WAM implementations for realizing the part of first-order unification that must be performed in interpretive mode.

While several data areas are carried over from the WAM, their usage in our machine differs in certain respects. In addition to storing compound terms that are created in a structure copying implementation, the heap is used in our context also to store disagreement pairs, the new terms that are generated during term reduction and the projection and imitation substitutions generated by higher-order unification. Similarly, the trail records not only the substitutions made for variables, but also the destructive changes made to terms during normalization and pointers to the pairs of terms removed from disagreement sets in the course of term simplification. Of particular note in this context are the facts that the trailing of terms requires also that old values be stored and that different kind of entries entail different unwinding actions and must be annotated appropriately for determining this. Finally, in addition to the usual choice point and environment records, the local stack must also store information about branch points in unification. These are distinguished by being labelled as *branch points* in Figure 4 that also depicts their split representation between a shared and a variable part. Only the variable parts of these records participate directly in the chain of backtracking records; the shared parts gain currency by being used by the variable parts. In the figure, we have used solid arrows to depict the shared part and the variable part of a branch point record and dashed arrows to depict the chain of branch points and choice points that determine backtracking behaviour.

The extended machine also includes a few new registers: the *LL* register indicating the currently active disagreement set, the *SL* register indicating the current top of the *SL* stack, the *BRS* register indicating the currently relevant shared part of branch point records and the *NUMARGS* register that holds the (current) arity of an application encountered during head normalization. One slightly intriguing aspect of our depiction of the machine state is the fact that the *S* register that indicates the argument vector of a compound term during

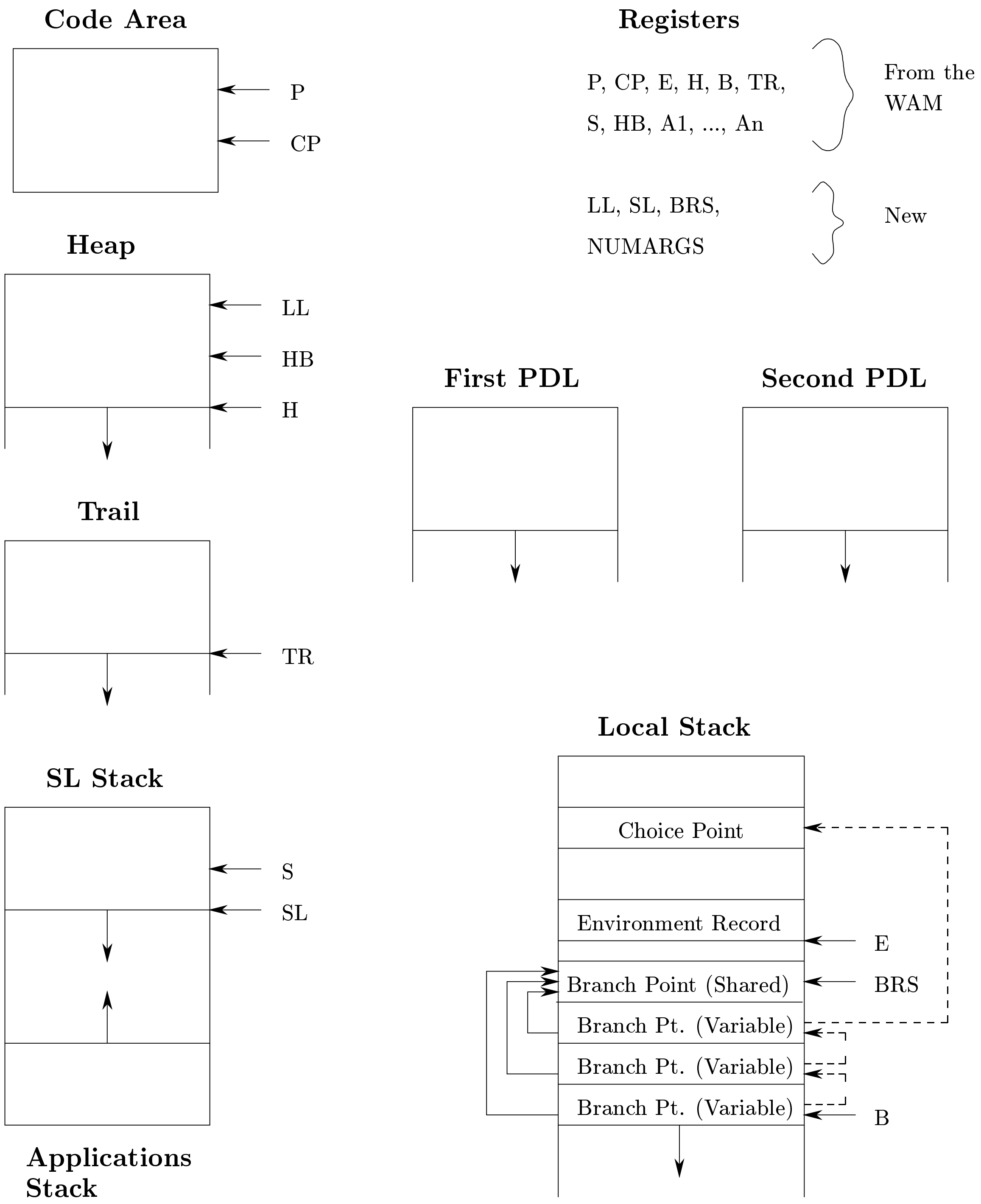

Figure 4: Abstract Machine Data Areas and Snapshot of State

unification is shown pointing into the *SL* stack rather than the heap. The reason for this is that the top-level, head-normalized structure of a higher-order term may become apparent only after a reduction process and, in this case, is available as a vector only in the *SL* stack. In special cases, such as when dealing with first-order terms, no reduction steps are necessary and our representation of such terms stores the arguments as a vector. Such situations can be recognized and, as an optimization, the *S* register can be made to point to the vector that is already available in the heap instead.

6.2 Modifications to the Instruction Set

A compilation model for our language must account for certain new aspects in comparison with the one for Prolog. These aspects include a representation of terms that differs even over the first-order fragment, the possibility for function variables and abstractions to appear in terms, the need to realize higher-order unification and the necessity to treat mixed intensional and extensional uses of predicate terms. We discuss these aspects in more detail below and we also indicate changes to the instruction set of the WAM that are geared towards treating them.

Creating Typed, Higher-Order Terms. The usual compilation model requires that the arguments of atomic goals appearing in the bodies of clauses be set up in registers prior to the invocation of code for the relevant procedures. In the case that such an argument is a compound term, its representation must be created in the heap with a reference to it being placed in the relevant register. These effects are actually realized through the *put* and *unify* classes of instructions present in the WAM, the latter being executed in write mode in these particular situations.

This basic structure carries over well to our higher-order language and many specific instructions from the WAM can even be retained for processing first-order like structure. There are, however, two exceptions. First, in our context, types are retained with variables and the instructions that create them must, for this reason, take an extra type argument. In particular, these instructions might take on the forms

put_variable Vi,Aj,type, and
unify_variable Vi,type

where *Vi* is either a permanent or temporary variable and *type* is a reference to the representation of a type. The second difference arises from the modified representation of a structure. We encode this as an application whose argument part is a pointer to a vector with a size matching the arity of the application. Moreover, in the general case, the 'function' part of the application could be different from a constant. In light of this, the *put_structure* instruction might be generalized to a *put_app* instruction that fashions an application on the heap. The abstract machine underlying the *Teyjus* system, in fact, includes two instructions of the form

put_capp Ai,Xj,n, and
put_fapp Ai,Xj,n

for this purpose. Each of these instructions creates an application whose function part is obtained from the register *Xj* and leaves a reference to this application in the register *Ai*. Moreover, the application that is created has arity n, a fact that is realized by allocating a vector of this size in the heap for the argument part and by preparing to fill in these arguments by setting the S register to the beginning of this vector and turning the write mode on. The difference between the two instructions is that the first annotates the application as closed whereas the second annotates it as (possibly) open.

The higher-order nature of our terms can manifest itself in three ways in the syntax: the function part may be a variable, abstraction may be explicitly present and there may be occurrences of abstracted variables. The instruction for creating applications already accounts for the special case of a variable 'functor.' To support the creation of abstractions, new instructions may be added to the *put* and *unify* classes. The abstract machine for *Teyjus* includes the following instructions for this purpose:

put_clambda Ai,Xj	*unify_clambda Xj*
put_flambda Ai,Xj	*unify_flambda Xj*

The *put* versions create abstractions whose bodies are given by the contents of register *Xj* on the heap and put references to these abstractions in the register *Ai*.[9] The difference between the two instructions provided for this purpose is that one creates closed abstractions and the other open ones. The *unify* versions, that are only ever executed in write mode, create similar abstractions but eventually put references to these in the heap location pointed to by the S register and also increment this register at the end. Finally, to support the creation of bound variables, represented using indices in the de Bruijn scheme, the following instructions in which n is a positive number, are included in our abstract machine:

put_index Ai,n	*unify_index n*

The first instruction writes a bound variable with index n on the heap and makes the register *Ai* a reference to this location. The *unify_index* instruction, which, also is only executed in write mode, stores this bound variable in the location pointed to by the S register and then increments this register.

In the instructions that create applications and abstractions, the function part and the abstraction body are both obtained from registers. However, these components may in particular situations correspond to permanent variables. Furthermore, they may actually dereference to stack cells that must be globalized prior to use. In light of these possibilities, our abstract machine includes the instructions *globalize Yi,Xj* and *globalize Xj*. The first instruction dereferences the permanent (environment) variable *Yi*. If this turns out to be a reference to the stack, then the value is copied to the heap and the stack cell and the register *Xj* are both coverted into references to the newly created heap cell. Otherwise the reference that we get to a heap cell is also stored in the register *Xj*. The second instruction simply

[9]The register *Xj* may contain a constant, in which case the first action of these instructions is to convert *Xj* into a reference to a location on heap containing this constant.

dereferences the *Xj* register, globalizes this as before if necessary and leaves a reference to a suitable heap cell in *Xj*.

Compilation of Higher-Order Unification. In any given use of a clause, the terms that appear as arguments of the head of a clause must be unified with the terms that arrive in the relevant argument registers. The compilation model for Prolog translates each of these statically known terms into a sequence of instructions that either creates a relevant term that the incoming argument is bound to if this argument is an uninstantiated variable and that carries out an analysis of the structure of the argument if it is not a variable. This model requires the same instructions to function in two different dynamically determined ways, an ability that is realized through the use of the read and write modes.

Lifting this treatment of unification to cover the operation in its entirety in the higher-order situation is difficult. In particular, statically available structure is not directly usable once a function variable with arguments is reached in it and is also difficult to exploit when a flexible, nonvariable part is exposed in the incoming term relative to which a set of matching substitutions have to be tried. However, at least the first phase of term simplification can be compiled and, if augmented with the simple forms of variable bindings discussed in Section 5.2, most of the unification computation that arises in practice can be treated completely within this phase.

The *get* and *unify* class of WAM instructions that treat head unification can, in fact, be adapted to realize this idea when the term to be compiled has a first-order structure at the top level, *i.e.*, when it is a variable, a constant or an application in which the head is a constant.[10] However, a few changes in interpretation are necessary for the instructions *get_structure*, *get_constant* and *unify_constant* that are used in compiling rigid structure. First, these instructions must take responsibility for head-normalizing the input term at the outset. In practice, many of these terms have a first-order structure, a fact that can be recognized through a few quick checks built into the relevant instructions so that an explicit invocation of head-normalization can be avoided. Notice that the *get_structure* instruction must set the *S* register to point to the vector of arguments in case the incoming term is itself an application with the right head and, under the considered optimization, this would become a pointer either into the *SL* stack or into the heap. The second change is that when the incoming term is a variable, the *get_structure* instruction must create an application of a specified arity on the heap and so should get this arity as an additional argument. In the *Teyjus* abstract machine, the instruction actually has the format

get_structure Ai,f,n

where n is a positive number; when executed in a mode in which a term has to be created, this instruction pushes an application with arity n and function part f onto the heap, followed by a vector of size n constituting the argument part of this application and sets the S register to the beginning of this vector. The final change arises from the fact that these instructions must also cater to the possibility that the incoming term is an application

[10] Since this term will be normalized prior to compilation, the only remaining possibilities are that it is an abstraction or an application with a variable at the head.

with a flexible head. One possible strategy in such a case is to add a suitable pair to the existing disagreement set and to leave its further processing to a later interpretive treatment of genuine higher-order unification. In the situation where the instruction is *get_structure*, we note that the added disagreement pair will actually involve a term that is created by subsequent actions carried out by this and following instructions executed in write mode.

It is in principle possible to extend the compilation of first-order like structure to include the case of terms that have abstractions at their head. However, it is not clear if enough situations where this is needed will occur in practice so as to make such a treatment pragmatically useful. We therefore describe a simpler approach that works uniformly for this case as well as for the last remaining case which is that of an application whose head is a variable. In essence, the term in both situations may be translated into a sequence of instructions that constructs its representation and leaves a reference to it in a register, followed by an instruction that invokes term simplification in interpretive mode. We have already discussed instructions for creating higher-order terms. To realize the last effect, we may use the *get_value* instruction from the WAM that, in any case, has to be adapted to deal with higher-order terms. In particular, in the new form, the instruction invokes an interpretive phase of term simplification that may make simple bindings for variables and that may add new flexible-rigid pairs to the existing disagreement set. A similar kind of generalization must be made to the *unify_value* instruction. Actually, another change to these instructions is also necessary. Although usual implementations of Prolog omit occurs-checks, the place to carry these out if they are included would be within the process invoked by the *get_value* and the *unify_value* instructions. The situation in the higher-order case is similar, except that rigid path checks would replace occurs checks. These checks turn out to be indispensable to the envisaged applications of the language whose implementation we are considering, and so they are included in the 'higher-order' versions of the *get_value* and *unify_value* instructions. Now, as discussed in Section 5.2, a rigid path check may force only a partial instantiation of a variable to be constructed in particular situations, with the rest of the instantiation being subject to the resolution of newly constructed disagreement pairs. In certain circumstances, the *unify_value* instruction may itself be required to write a variable to the heap and, for this reason, it acquires an additional type argument.

After simplification has been carried out relative to all the terms appearing in the head of a clause, it may be necessary to invoke an interpretive phase of higher-order unification. Our abstract machine includes three instructions for this purpose. One of these, the *proceed_finish_unify* instruction, is used in place of the *proceed* instruction of the WAM in the situation when the clause body is empty and when an unresolved higher-order unification problem may exist. The effect of this instruction is to set the program pointer to the continuation point, to set up the shared part of a branch point record and, finally, to invoke code that tries to complete the unification process. The code that is invoked tries to generate a matching substitution. If one is found, then this is applied to the state, the variable part of a branch point record representing the remaining matching substitutions is created and the simplification and substitution generation processes are iterated. A point to note about the situation in which *proceed_finish_unify* is used is that no argument registers need to be stored in the shared part of the branch point record. The second instruction, *execute_finish_unify*, is used when the body of the clause consists of a single atomic goal. This instruction differs from *proceed_finish_unify* in that it must update the program pointer to the next instruction in sequence and also save the continuation point and relevant argument registers in the shared part of the branch point record for use on backtracking. The number of argument registers that must be remembered becomes a parameter to this instruction. The final instruction, *call_finish_unify*, is used when the body of the clause has multiple goals in it, and therefore requires an environment record to be created for its invocation. This instruction behaves differently from *execute_finish_unify* in only two respects. First, it does not need to save the continuation point since this is available from the environment record. Second, before it allocates space for the shared part of the branch point record, the instruction must ensure that sufficient space has been left for the permanent variables in the clause. On account of the latter requirement, *call_finish_unify* acquires the count of the permanent variables as an argument, in addition to the count of the register arguments that need to be saved.

The interpretive phase of unification is, of course, not always needed. In particular, it need only be considered if the the compiled form of term simplification leads to additions to the original disagreement set or to bindings for variables that have the potential of modifying the status of existing pairs. Static checks can sometimes be used to determine whether this happens and, consequently, to decide if the new instructions must be used. Even when the instructions are included in the compiled code, simple dynamic checks can be incorporated within them to quickly decide the irrelevance of further processing in certain instances. The *Teyjus* implementation utilizes such ideas to avoid unnecessary examination of disagreement sets and setting up of the shared parts of branch point records.

Treating Mixed Uses of Predicates. The crux of this treatment is the compilation of flexible atomic goals: mixed uses of predicate terms arises essentially from the fact that flexible goals may be instantiated by terms with complex logical structure, thereby reflecting intensional occurrences of quantifiers and connectives into positions where they function as search directives.

The problem in the treatment of flexible atomic goals is, of course, that their top-level structure is determined dynamically, and so the specific action to be performed is not known at compilation time. Nevertheless, some part of the action can be compiled by using the knowledge of the possible cases that can arise. In particular, flexible goals can be compiled into calls to a special procedure named *solve* to which (the instantiated version of) the goal is provided as an argument. In the case that (the normalized form of) the instantiated goal has a complex structure, the behavior of *solve* can be envisaged as if it were based on a compilation of the following clauses in which we use semicolon to represent disjunction in an extensional position:

$solve\ (G1 \wedge G2)$:- $(solve\ G1), (solve\ G2)$.
$solve\ (G1 \vee G2)$:- $(solve\ G1); (solve\ G2)$.
$solve\ (\Sigma\ G)$:- $solve\ (G\ X)$.

To complete the description of *solve*, it only remains to specify its behaviour in the situation when its argument is an atomic goal. In the case that this goal is a flexible one, *solve* succeeds after instantiating the head of the goal to a term of the form $\lambda \ldots \lambda \top$, the binder being chosen based on type considerations. If this goal is a rigid one, then its arguments are loaded into appropriate argument registers and the head is used to determine the code to be invoked next.

In the *Teyjus* implementation, the *solve* predicate is treated as a builtin one and its realization is 'hard-wired' into that of the abstract machine.

6.3 Examples of Compiled Code

Based on the compilation scheme described in this section, code of the following form might be generated from the definition of the *mapfun* predicate presented in Section 2:

```
mapfun:  switch_on_term L2, L3, L5, fail   %
L2:      try_me_else L4, 3                 % mapfun
L3:      get_nil A1                        % nil
         get_nil A3                        % F nil
         proceed_finish_unify              %
L4:      trust_me 3                        % mapfun
L5:      get_list A1                       % (::
         unify_variable A4, ty1            % X
         unify_variable A1, ty2            % L1)
         get_list A3                       % F (::
         unify_variable A5, ty1            % S1
         unify_variable A3, ty2            % L2)
         globalize A2                      %
         put_capp A6, A2, 1                % S2 = (F
         unify_value A4, ty1               % X)
         get_value A6, A5                  % S1 = S2
         execute_finish_unify 3            % :-
         execute mapfun                    % mapfun L1 F L2
```

This code uses the instructions *get_nil* and *get_list* that realize, as in the WAM, special cases of the *get_constant* and *get_structure* instructions. Also used is the instruction *switch_on_term* that adapts an indexing instruction with the same name from the WAM. In our context, this instruction takes the form

switch_on_term V,C,L,BV

where V, C, L and BV are addresses to which control must be transferred in case the dereferenced and head-normalized version of the value stored in register *A1* is, respectively, a flexible term, a rigid term that has a constant different from *::* as its head, a nonempty list or a term with a bound variable as its head. In the use that is made of this instruction above, *fail* is assumed to be the location of code that causes backtracking. The instructions *try_me_else* and *trust_me* that are used here function as they do in the WAM to create, utilize and discard choice points; an extra numeric argument has been included with each of them that indicates the number of argument registers that are to be saved or retrieved as relevant. The *unify_variable* and *unify_value* instructions that are used take type parameters for reasons that we have already explained. In this particular instance, *ty1* and *ty2* are to be understood as references to the representation of the types i and *(list i)*, respectively. We note that in the only place where the *unify_value* instruction appears in this code, there is no utility for the type argument and, observing that this instruction will never be executed in read mode, we may replace it with a special *set_value* instruction as suggested in [3].[11] As a final comment, we observe that both the *proceed_finish_unify* and the *execute_finish_unify* instructions that appear in this code are essential: depending on the form of the first and third incoming arguments, execution of the term simplification code for either clause may lead to bindings that affect the state of the existing disagreement set.

The definition of *mappred* presented in Section 2 illustrates a mixed use of a predicate variable. Compilation of that definition might produce the following code:

```
mappred:  switch_on_term L2, L3, L5, fail   %
L2:       try_me_else L4, 3                 % mappred
L3:       get_nil A1                        % nil
          get_nil A3                        % P nil
          proceed_finish_unify              %
L4:       trust_me 3                        % mappred
L5:       allocate                          %
          get_list A1                       % (::
          unify_variable A4, ty1            % X
          unify_variable Y2, ty2            % L1)
          get_variable Y1, A2               % P
          get_list A3                       % (::
          unify_variable A2, ty1            % Y
          unify_variable Y3, ty2            % L2)
          call_finish_unify 3, 3            % :-
          globalize Y1, A3                  %
          put_capp A1, A3, 2                % S1 = (P
          unify_value A4, ty1               % X
          unify_value A2, ty1               % Y)
          call solve, 3                     % S1,
          put_value Y2, A1                  % (mappred L1
          put_value Y1, A2                  % P
          put_value Y3, A3                  % L2
          deallocate                        %
```

[11]This is, in fact, what is done in the abstract machine and compilation model actually underlying the *Teyjus* implementation.

execute mappred %)

We assume here that *ty1* and *ty2* are references to the representation of the types i and *(list i)*, respectively. The flexible goal *(P X Y)* is translated in this code by a call to the predicate *solve* as discussed earlier in this section. Towards understanding the nature of this translation, we might consider the execution of the query

$$mappred\ (bob :: sue :: nil)\ (\lambda x\, \lambda y\, \exists z\, (parent\ x\ z) \wedge (parent\ z\ y))\ L.$$

discussed in Section 2. Clause indexing will lead to the selection of the code for the second clause for *mappred* in this case. The term simplification part of this code will execute successfully, the term $\exists z\, (parent\ bob\ z) \wedge (parent\ z\ y))$ will be formed and stored in register *A1* and the code for *solve* will be invoked. Using the definition of *solve*, this goal will be simplified, leading eventually to the invocation of the atomic goals *(parent bob Z)* and *(parent Z Y)*. The recursive call to *mappred* will lead, in a similar fashion, to the invocation of the atomic goals *(parent sue Z')* and *(parent Z' Y')*. The query variable L will be bound at the end to a list containing the values determined for Z and *Z'* by these goals. Another point to note is that all the unification problems that arise relative to the query of interest are ones that can be solved without the invocation of the interpretive, higher-order phase.

7 Conclusion

We have considered in this paper the implementation of an extension to logic programming that is based on permitting a quantification over predicate and function symbols and on using lambda terms as data structures in place of first-order terms. In addition to a careful exposition of the issues that need to be dealt with in a low-level realization of such an extension, our contributions are threefold: we have discussed representations for lambda terms that facilitate their intensional treatment, we have presented mechanisms for realizing term reduction and for supporting higher-order unification within a logic programming machine model and we have sketched an approach to compilation. The ideas that we have presented here have been used in amalgamation with other devices that we have developed for the treatment of new scoping mechanisms and of polymorphic typing in an actual implementation of the λProlog language.

A question often of interest in the context of language enrichments is the performance degradation that is to be incurred on account of them. There are two factors that lead to a different treatment of first-order programs within our framework from that in traditional Prolog implementations. First, as discussed in Section 4.5, a representation must be used for compound terms that permits changes to be made to internal nodes in their tree-like structure. Second, the occurs-check that is usually omitted in logic programming languages is not really a luxury in the important higher-order applications. A third factor, not discussed here but that is relevant to the full λProlog language, is a runtime overhead arising from polymorphic typing. The impact of the occurs-check is obviously non-uniform and therefore impossible to quantify in a general manner. A careful assessment of the first and

System	*Special List Representation (Polymorphic)*	*Functor Based Representation*	
		Monomorphic	*Polymorphic*
Teyjus (v 1.0-b32)	11.99 secs	18.67 secs	21.18 secs
SWI-Prolog (v 4.0.0)	8.1 secs	8.8 secs	
SICStus (v 3.9.1)	0.23 secs	0.35 secs	

Figure 5: Timing Comparisons over Naive Reverse

third factors requires experiments with controlled auxiliary implementations, something beyond the scope of this paper. However, a rough assessment is possible. Lists receive a specialized treatment in the *Teyjus* system that comes close to the usual representation of first-order structures. By contrasting performance under such a treatment with that when a vanilla functor-based representation is used, a sense of the additional cost can be obtained.

Figure 5 presents the results of the kind of experiment described above, performed with *Teyjus* version 1.0-b32 modified to omit the occurs-check. The numbers in the table represent the time taken by 10,000 invocations of naive reverse on a 30 element list. All trials, here and below, were carried out on a 440 MHz UltraSPARC-IIi processor. A functor-based representation for lists in *Teyjus* can be chosen to be either monomorphic or polymorphic in nature and execution times are provided for both. In contrast, the specialized list representation is available only in polymorphic form. From the timing measurements for the polymorphic versions, we conclude that there is about a 75% overhead to not using the specialized representation. This is appropriately viewed as an upper bound on the additional cost for a higher-order representation, at least some of the improved performance being attributable to specialized compilation for lists. Polymorphism adds about a 13.5% overhead in the functor-based representation and we estimate a similar cost under the special treatment of lists. For comparison, we also present performance measurements for two Prolog implementations; from the perspective of running time, these figures are best thought of as applying to monomorphic list representations. The contrast with *SICStus* is humbling, indicating the distance to go in building a well-engineered and highly optimized implementation, even if revealing little by way of the difference between treatments of the first-order and the higher-order language.

Another important aspect of comparison is that of contrasting our ideas and system with those of other implementations of λProlog. There have been four previous implementations of this language. Three of these are interpreter based, built using Prolog [28], Lisp [12] and Standard ML [13, 45]. None of these systems considered in any detail the special issues that arise in a low-level treatment of the higher-order aspects of λProlog and a comparison with them therefore appears not to be very meaningful.[12] The only remaining realization of λProlog, called *Prolog/Mali* [8], is one that translates λProlog programs into C code that can then be compiled. The translation process utilizes a memory management system called

[12]The performance comparisons made in [8] with the Lisp version substantiate this viewpoint.

System	*Naive Reverse*	*Type Inference*
Teyjus (v 1.0-b32)	11.99 secs	2.95 secs
Prolog/Mali	12.00 secs	9.59 secs

Figure 6: Comparisons between Teyjus and Prolog/Mali

Mali that has been developed especially for logic programming languages: in particular, translation is realized in the form of calls to functions supported by this system. Using this approach has the distinct benefit that a memory management scheme is automatically available but it also forces some awkward choices, such as the full copying of clause bodies, to be in consonance with the framework provided by *Mali*.

Despite the difference in overall structure, there is a scheme to the treatment of the higher-order aspects in *Prolog/Mali* that can be compared with the ideas we have presented in this paper. At the level of term representation, there seem to be three differences. First, the de Bruijn scheme for rendering bound variables is rejected in *Prolog/Mali* on the grounds that "it forces to renumber the rightmost term." While this observation is correct in principle, it appears not to be relevant in practice as we have pointed out in Section 4.1. To support the comparison of terms in a situation where a name-based encoding is used for bound variables, an approach based on using new constants is suggested. Unfortunately, the details of this approach are not explained completely making a satisfactory assessment of it impossible.[13] A second difference is that an explicit substitution mechanism is not considered in *Prolog/Mali* and reduction substitutions seem to be effected eagerly. Finally, first-order terms seem to obtain the usual Prolog-like treatment in *Prolog/Mali*, higher-order facets being handled via special attributes attached to terms. The treatment of higher-order unification and the integration of reduction into the overall computational model receives little discussion in [8] and, in light of this, we believe that a detailed consideration of these aspects is unique to our work; an interesting exception, however, is the idea of indexing flexible-flexible pairs by their flexible heads, to be awakened by bindings for these heads, a possibility whose integration into our processing model bears investigation. The last relevant aspect is the compilation of unification. Clearly, the underlying machine model is explicitly manifest only in our work although many ideas relating to the compilation of the first phase of simplification of disagreement sets receive a similar treatment in both contexts and and share also with an early presentation of some of our ideas [32].

Figure 6 complements our qualitative comparisons by presenting execution times for *Prolog/Mali* and *Teyjus* on two different kinds of tasks. The *naive reverse* program is the one used in the earlier tests and, as such, provides a measure of behaviour over first-order programs. The *type inference* program assigns type schemes to ML-like programs and is a good example for testing performance over higher-order terms, reduction and (a specialized form of) higher-order unification. The indications from these tests is that

[13] There are also vestiges of this approach in answer presentations that remain unclear to us and, quite possibly, to other λProlog users.

the *Teyjus* system matches performance of *Prolog/Mali* over first-order programs and does substantially better on genuine higher-order ones. A larger set of tests is needed to draw more substantive conclusions. Unfortunately, there are practical difficulties to providing a suitable collection that is indicative of genuine performance differences. *Prolog/Mali* omits the occurs-check that is significant to higher-order applications, uses a non-standard syntax for λProlog programs leading to a substantial overhead in adapting available user programs to run under it and, finally, appears to yield incorrect results in a few of the examples tried.

Our focus in this paper has been on describing a broad framework for the treatment of higher-order features in logic programming. There are obviously tradeoffs in the actual deployment of these ideas. Although beyond the scope of the present study, a quantification of these tradeoffs is important and is, in fact, the object of ongoing work (*e.g.*, see [23]). A particularly exciting direction that we are now exploring is that of fine-tuning our abstract machine and compilation model to the important subclass of higher-order programs referred to as L_λ programs [26], possibly even with some loss of completeness over the full collection. In a different vein, many of our implementation ideas are applicable in related contexts, such as that of logic programming within a dependently typed lambda calculus [42]. The extension of this work in these directions is also a matter under investigation.

Acknowledgements

Earlier versions of the ideas in this paper have benefitted from discussions with Bharat Jayaraman and Debra Wilson. This research has been supported by the National Science Foundation under Grant No. 9803849 and and Grant No. 0096322.

References

[1] Martín Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, 1991.

[2] L. Aiello and G. Prini. An efficient interpreter for the lambda-calculus. *The Journal of Computer and System Sciences*, 23:383–425, 1981.

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

[4] Peter B. Andrews. Resolution in type theory. *Journal of Symbolic Logic*, 36:414–432, 1971.

[5] A. W. Appel and A. P. Felty. Lightweight lemmas in λProlog. In *International Conference on Logic Programming*, pages 411–425. MIT Press, November 1999.

[6] K. R. Apt and M. H. van Emden. Contributions to the theory of logic programming. *Journal of the ACM*, 29(3):841–862, 1982.

[7] Z. Benaissa, D. Briaud, P. Lescanne, and J. Rouyer-Degli. $\lambda\upsilon$, a calculus of explicit substitutions which preserves strong normalization. *Journal of Functional Programming*, 6(5):699–722, 1996.

[8] Pascal Brisset and Olivier Ridoux. The compilation of λProlog and its execution with MALI. Publication Interne No 687, IRISA, Rennes, November 1992.

[9] N. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser Theorem. *Indag. Math.*, 34(5):381–392, 1972.

[10] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[11] G. Cousineau, P-L. Curien, and M. Mauny. The categorical abstract machine. *The Science of Programming*, 8(2):173–202, 1987.

[12] Conal Elliott and Frank Pfenning. eLP, a Common Lisp Implementation of λProlog. Implemented as part of the CMU ERGO project, May 1989.

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

[14] J. Fairbairn and S. Wray. TIM: A simple lazy abstract-machine to execute super-combinators. In G. Kahn, editor, *Functional Programming Languages and Computer Architecture*, number 274 in Lecture Notes in Computer Science, pages 34–45. Springer-Verlag, 1987.

[15] Amy Felty. Implementing tactics and tacticals in a higher-order logic programming language. *Journal of Automated Reasoning*, 11(1):43–81, August 1993.

[16] John Field. On laziness and optimality in lambda interpreters: Tools for specification and analysis. In *Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 1–15. ACM Press, January 1990.

[17] John Hannan and Dale Miller. From operational semantics to abstract machines. *Mathematical Structures in Computer Science*, 2(4):415–459, 1992.

[18] John Hannan and Frank Pfenning. Compiler verification in LF. In *Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 407–418. IEEE Computer Society Press, June 1992.

[19] Peter Henderson and James H. Morris. A lazy evaluator. In *Third Annual ACM Symposium on Principles of Programming Languages*, pages 95–103, 1976.

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

[21] Fairouz Kamareddine and Alejandro Ríos. Extending the λ-calculus with explicit substitution which preserves strong normalization into a confluent calculus on open terms. *Journal of Functional Programming*, 7(4):395–420, 1997.

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

[23] Chuck Liang and Gopalan Nadathur. Tradeoffs in the intensional representation of lambda terms. In S. Tison, editor, *Conference on Rewriting Techniques and Applications*, Lecture Notes in Computer Science, pages 192–206. Springer-Verlag, July 2002.

[24] Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Transactions on Programming Lanuages and Systems*, 4(2):258–282, April 1982.

[25] Spiro Michaylov and Frank 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.

[26] Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1(4):497–536, 1991.

[27] 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. IEEE Computer Society Press, September 1987.

[28] Dale Miller and Gopalan Nadathur. λProlog version 2.7. Distributed in C-Prolog and Quintus Prolog source code, August 1988.

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

[30] Gopalan Nadathur. An explicit substitution notation in a λProlog implementation. Technical Report TR-98-01, Department of Computer Science, University of Chicago, January 1998.

[31] Gopalan Nadathur. A fine-grained notation for lambda terms and its use in intensional operations. *Journal of Functional and Logic Programming*, 1999(2), March 1999.

[32] 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. MIT Press, October 1989.

[33] Gopalan Nadathur, Bharat Jayaraman, and Keehang Kwon. Scoping constructs in logic programming: Implementation problems and their solution. *Journal of Logic Programming*, 25(2):119–161, November 1995.

[34] 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. MIT Press, August 1988.

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

[36] Gopalan Nadathur and Dale Miller. Higher-order logic programming. In C. Hogger D. Gabbay and A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 5, pages 499–590. Oxford University Press, 1998.

[37] Gopalan Nadathur and Dustin J. Mitchell. System description: Teyjus—a compiler and abstract machine based implementation of λProlog. In Harald Ganzinger, editor, *Automated Deduction–CADE-16*, number 1632 in Lecture Notes in Artificial Intelligence, pages 287–291. Springer-Verlag, July 1999.

[38] 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.

[39] Gopalan Nadathur and Debra Sue Wilson. A notation for lambda terms: A generalization of environments. *Theoretical Computer Science*, 198(1-2):49–98, 1998.

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

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

[42] Frank Pfenning and Carsten Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, pages 202–206, Trento, Italy, July 1999. Springer-Verlag LNAI 1632.

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

[44] D.H.D. Warren. Implementing Prolog—compiling predicate logic programs. University of Edinburgh, D.A.I Research Reports 39 and 40, May 1977.

[45] Philip Wickline and Dale Miller. The Terzo 1.1b implementation of λProlog. Distribution in NJ-SML source files. See `http://www.cse.psu.edu/~dale/lProlog/`, April 1997.