# The Metalanguage λProlog and Its Implementation

Gopalan Nadathur

Department of Computer Science and Engineering
University of Minnesota
4-192 EE/CS Building, 200 Union Street SE
Minneapolis, MN 55455
`gopalan@cs.umn.edu`
Home Page: `http://www.cs.umn.edu/~gopalan`

**Abstract.** Stimulated by concerns of software certification especially as it relates to mobile code, formal structures such as specifications and proofs are beginning to play an explicit role in computing. In representing and manipulating such structures, an approach is needed that pays attention to the binding operation that is present in them. The language λProlog provides programming support for a higher-order treatment of abstract syntax that is especially suited to this task. This support is realized by enhancing the traditional strength of logic programming in the metalanguage realm with an ability for dealing directly with binding structure. This paper identifies the features of λProlog that endow it with such a capability, illustrates their use and and describes methods for their implementation. Also discussed is a new realization of λProlog called *Teyjus* that incorporates the implementation ideas presented.

## 1 Introduction

The language λProlog is based on the higher-order theory of hereditary Harrop formulas that embodies a rich interpretation of the abstract idea of logic programming [18]. Through a systematic exploitation of features present in the underlying logic, this language realizes several capabilities at the programming level such as ones for typing, scoping over names and procedure definitions, representing and manipulating complex formal structures, modularly constructing code and higher-order programming. Our interest in this paper is in one specific facet of λProlog: its role as a metalanguage.

The manipulation of symbolic expressions has been of longstanding interest and some of the earliest computational tasks to have been considered and systematically addressed have, in fact, concerned the realization of reasoning processes, the processing of human languages and the compilation and interpretation of programming languages. The calculations involved in these cases are typically metalinguistic and syntactic in nature and a careful study of their structure has produced a universally accepted set of concepts and tools relevant to this form of computing. An important component in this collection is the idea of *abstract syntax* that moves away from concrete presentation and focuses instead on the essential relationships between the constituent parts of symbolic constructs. A complementary development has been that of languages that provide programming support for computing with abstract syntax. These languages, which include Lisp, ML and Prolog amongst them, contain mechanisms that simplify the representation, construction and deconstruction of abstract syntax and that permit the implicit management of space relative to such manipulations. Effort has also been invested in implementing these languages efficiently, thereby making them practical vehicles for realizing complex symbolic systems.

One may wonder against this backdrop if anything new really needs to be added to the capabilities already available for symbolic computation. The answer to this question revolves around the treatment of scope and binding. Many symbolic objects whose manipulation is of interest involve forms of these operations in their structure in addition to the compositionality that is traditionally treated in abstract syntax. This is true, for instance, of quantified formulas that are considered within reasoning systems and of procedures with arguments that are of interest to programming language compilers. The conventional approach in these cases has been to use auxiliary mechanisms to avoid explicit reference to binding in representation. Thus, reasoning systems eliminate quantifiers from formulas through a preprocessing phase and compilers utilize symbol tables to create binding environments when these are needed in the analysis of programs. While such methods have been successful in the past, there is now an increasing interest in formal constructs with sophisticated and diverse forms of scope whose uniform treatment requires a reflection of the binding operation into abstract syntax itself. The desire to reason in systems different from classical logic provides one example of this kind. The elimination of quantifiers may either not be possible or desirable in many of these cases, requiring them to be explicitly represented and dynamically treated by the reasoning process. In a similar vein, motivated by the proof-carrying-code approach to software certification [29], attention has been paid to the representation of proofs. The discharge of assumptions and the treatment of genericity are intrinsic to these formal structures and a convenient method for representing such operations involves the use of binding constructs that range over their subparts. As a final example, relationships between declarations and uses are an important part of program structure and a formal treatment of these in representation can influence new approaches to program analysis and transformation.

Driven by considerations such as these, much effort has recently been devoted to developing an explicit treatment of binding in syntax representation, culminating in what has come to be known as *higher-order abstract syntax* [31]. The main novelty of $\lambda$Prolog as a metalanguage lies in the support it offers for this new approach to encoding syntactic objects. It realizes this support by enriching a conventional logic programming language in three essential ways. First, it replaces first-order terms—the data structures of a logic programming language—by the terms of a typed lambda calculus. Attendant on these lambda terms is a notion of equality given by the $\alpha$-, $\beta$- and $\eta$-conversion rules. The main difference in representational power between first-order terms and lambda terms is that the latter are capable of also capturing binding structure in a logically precise way. Thus, this enhancement in term structure endows λProlog with a means for representing higher-order abstract syntax. Second, λProlog uses a unification operation that builds in the extended notion of equality accompanying lambda terms. This change provides the language with a destructuring operation that can utilize information about binding structure. Finally, the language incorporates two new kinds of goals, these being expressions of the form $\forall x G$ and $D$ `=>` $G$, in which $G$ is a goal and $D$ is a conjunction of clauses.[1] A goal of the form $\forall x G$ is solved by replacing all free occurrences of $x$ in $G$ with a new constant and then solving the result and a goal of the form $D$ `=>` $G$ is solved by enhancing the existing program with the clauses in $D$ and then attempting to solve $G$. Thus, at a programming level, the new forms of goals, which are referred to as *generic* and *augment*, respectively, provide mechanisms for scoping over names and code. As we shall see presently, these scoping abilities can be used to realize recursion over binding structure.

Our objective in this paper is to show that the new features present in λProlog can simplify the programming of syntax manipulations and that they can be implemented with sufficient efficiency to be practical tools in this realm. Towards this end, we first motivate the programming uses of these features and then discuss the problems and approaches to realizing them in an actual system. The ideas we discuss here have been used in a recent implementation of λProlog called *Teyjus* [25] that we also briefly describe. We assume a basic familiarity with lambda calculus notions and logic programming languages and the methods for implementing them that are embedded, for instance, in the Warren Abstract Machine (WAM) [35]. Further, in keeping with the expository nature of the paper, we favor an informal style of presentation; all the desired formality can be found in references that are cited at relevant places.

## 2 Higher-Order Abstract Syntax in λProlog

A common refrain in symbolic computation is to focus on the essential functional structure of objects. This is true, for instance, of systems that manipulate programs. Thus, a compiler or interpreter that manipulates an expression of the form *if B then T else E* must recognize that this expression denotes a conditional involving three constituents: $B$, $T$ and $E$. Similarly, a theorem prover that

[1] To recall terminology, a goal is what appears in the body of a procedure or as a top level query and is conventionally formed from atomic goals via conjunction, disjunction and existential quantification. Clauses correspond to procedure definitions. While a free variable in a clause is usually assumed to be implicitly universally quantified at the head of the clause, there is ambiguity about the scope and force of such quantification when the clause appears in an expression of the form $D$ `=>` $G$. λProlog interprets the scope in this case to be the entire expression of which $D$ `=>` $G$ itself may only be a part, and it bases the force on whether this expression is a goal or a clause. All other interpretations need to be indicated through explicit quantification.

encounters the formula $P \wedge Q$, must realize that this is one representing the conjunction of $P$ and $Q$. Conversely, assuming that we are not interested in issues of presentation, these are the *only* properties that needs to be recognized and represented in each case. The 'abstract syntax' of these expressions may therefore be captured by the expressions *cond(B,T,E)* and *and(P,Q)*, where *cond* and *and* are suitably chosen function symbols or constructors.

Another important idea in syntax based computations is that of structural operational semantics that advocates the description of computational content through rules that operate on abstract syntax. For example, using $\rhd$ as an infix notation for the evaluation relation, the operational meaning of a conditional expression can be described through the rules

$$\frac{B \rhd \mathit{true} \qquad T \rhd V}{\mathit{cond}(B,T,E) \rhd V}$$

$$\frac{B \rhd \mathit{false} \qquad E \rhd V}{\mathit{cond}(B,T,E) \rhd V}$$

Similarly, assuming that $\Gamma \longrightarrow F$ represents the judgement that $F$ follows from a set of assumptions $\Gamma$, the logical content of a conjunction can be captured in the rule

$$\frac{\Gamma \longrightarrow P \qquad \Gamma \longrightarrow Q}{\Gamma \longrightarrow \mathit{and}(P,Q)}$$

Rules such as these can be used in combination with some control regimen determining their order of application to actually evaluate programs or to realize reasoning processes.

The appropriateness of a logic programming language for symbolic computation arises from the fact that it provides natural expression to both abstract syntax and rule based specifications. Thus, expressions of the form *cond(B,T,E)* and *and(P,Q)* are directly representable in such a language, being first-order terms. Depending on what they are being matched with, the unification operation relative to these terms provides a means for constructing, deconstructing or recognizing patterns in abstract syntax. Structural operational rules translate directly to program clauses. The evaluation rules for conditional expressions can, for instance, be represented by the clauses

```
eval(cond(B,T,E),V) :- eval(B,true), eval(T,V).
eval(cond(B,T,E),V) :- eval(B,false), eval(E,V).
```

Using these rules to realize interpretation may require capturing additional control information, but this can be done through the usual programming devices.

## 2.1 The Explicit Representation of Binding

Many syntactic objects involve a form of binding and it may sometimes be necessary to reflect this explicitly in their representation. Binding structure can be represented only in an approximate manner using conventional abstract syntax or first order terms. For example, consider the formula $\forall x P(x)$. This formula may be represented by the expression $all(x, P(x))$. However, this representation misses important characteristics of quantification. Thus, the equivalence of $\forall x P(x)$ and $\forall y P(y)$ is not immediately present in the 'first-order' rendition and has to be built in through auxiliary processes. In a related sense, suppose it is necessary to instantiate the outer quantifier in the formula $\forall x \exists y P(x, y)$ with the term $t(y)$. The renaming required in carrying out this operation has to be explicitly programmed under the indicated representation.

The availability of lambda terms in $\lambda$Prolog provides a different method for dealing with these issues. A binding operator has two different characteristics: it determines a scope and it identifies a particular kind of term. In $\lambda$Prolog, the latter role may be captured by a suitably chosen constructor while the effect of scope may be reflected into a (metalanguage) abstraction. This form of representation is one of the main components of the higher-order approach to abstract syntax. Using this approach, the formula $\forall x P(x)$ might be rendered into the term *(all* $\lambda x$*(P x))*, where *all* is a constructor chosen to represent the predicative force of the universal quantifier; we employ an infix, curried notation for application here and below as is customary for higher-order languages, but the correspondence to the first-order syntax should be evident. Similarly, the program fragment

*lambda (x) if (x = 0) then (x - 2) else (2 * x)*

in a Lisp-like language might be represented by the term

*(abs* $\lambda x$*(cond (eq x 0) (minus x 2) (times 2 x)))*

where *abs* is a constructor that identifies an object language abstraction and *eq*, *plus*, *minus*, *0*, and *2* are constructors corresponding to the relevant programming language primitives. As a final, more involve example, consider the following code in a functional programming language:

*fact m n = if (m = 0) then n else (fact (m - 1) (m * n))*

This code identifies *fact* as a function of two arguments that is defined through a fixed point construction. Towards making this structure explicit, the given program fragment may be rewritten as

```
fact = (fixpt (f) (lambda (m) lambda (n)
                  if (m = 0) then n else (f (m - 1) (m * n))))
```

assuming that *fixpt* represents a binding operator akin to *lambda*. Now, using the constructor *fix* to represent this operator and *app* to represent object language application, the expression that is identified with *fact* may be rendered into the following $\lambda$Prolog term:[2]

[2] We are taking liberties with $\lambda$Prolog syntax here: the language employs a different notation for abstraction and all expressions in it are typed. In a more precise presen-

*(fix $\lambda f$(abs $\lambda m$(abs $\lambda n$*
*(cond (eq m 0) n (app (app f (minus m 1)) (times m n)))))).*

The higher-order abstract syntax representation of binding structure solves the problems that were discussed relative to the first-order encoding. The formulas $\forall x P(x)$ and $\forall y P(y)$ translate to *(all $\lambda x$(P x))* and *(all $\lambda y$(P y))*, but these are really the same terms by virtue of the understanding of $\alpha$-conversion present in the metalanguage. Similarly, the instantiation of the quantifier in a formula represented by *(all P)* with the term represented by $t$ is given simply by *(P t)*; the correct substitution, with all the necessary renaming operations, is realized from this through the $\beta$-conversion rule. The real power of this approach arises from the fact that the *same* principles apply to many other situations where binding is present. The encoding of programs, for instance, manifests an insensitivity to the names of function arguments by virtue of the same $\alpha$-conversion rule. Alternatively, consider the task of evaluating functional programs. Using the notation *F[f:=T]* to depict the logically correct substitution of $T$ for $f$ in $F$, one of the rules relevant to this is the following:

$$\frac{F[f{:=}(\mathit{fixpt}\ (f)\ F)] \rhd V}{(\mathit{fixpt}\ (f)\ F) \rhd V}$$

This rule can be encoded in the following $\lambda$Prolog clause:

*(eval (fix F) V)* `:-` *(eval (F (fix F)) V).*

The required substitution is, once again, realized via the $\beta$-conversion rule.

## 2.2 Structure Analysis using Higher-Order Unification

Another useful property of the higher-order abstract syntax representation is that the unification operation over it provides for sophisticated forms of structure analysis. This observation has been used previously by Huet and Lang in recognizing program properties [9]. Consider, for example, the term

*(fix $\lambda f$(abs $\lambda m$(abs $\lambda n$*
*(cond (C m n) (T m n) (app (app f (E1 m n)) (E2 m n))))))*

in which the symbols *C*, *T*, *E1* and *E2* represent variables that may be instantiated to obtain terms that correspond to actual programs. Thus, the program term corresponding to *fact* is obtained from this term through the substitution of $\lambda m \lambda n$*(eq m 0)*, $\lambda m \lambda n$ *n*, $\lambda m \lambda n$*(minus m 1)* and $\lambda m \lambda n$*(times m n)* for these respective variables. However, the logic places a restriction on what constitute correct instantiations: these cannot be carried out by terms that contain variables that get bound by the abstractions pertaining to *f*, *m* or *n*. Any dependencies in the subparts governed by *C*, *T*, *E1* and *E2* on the enclosing abstractions must,

tation the user would, for instance, identify a new sort *tm* to correspond to program terms and *fix* and *abs* would be given the types *(tm $\to$ tm) $\to$ tm*. We elide these aspects for paucity of space, referring the reader to, e.g., [24] for such details.

therefore, be realized through the arguments of these variables. As a consequence of this requirement, the abstracted variable *f* must appear in exactly one place in any program term that matches with the 'template' being considered—as the head of the right arm of the conditional. It is easy to see that any program that corresponds to such a term must be tail recursive. The displayed term functions, in this sense, as a recognizer for tail recursive programs.

Unfortunately, the template displayed is very limited in its applicability: any program it matches with must have a conditional as a body, must not contain nested conditionals, must have no recursive calls in the left branch of the conditional and must have a right branch that consists entirely of a recursive call. There are programs that violate all these requirements while still being tail recursive. A more important observation is that the limitation is inherent in any recognition scheme that uses templates alone: since conditionals can be arbitrarily nested, no finite collection of templates can be provided that recognize all tail recursive programs and only these. However, there is a recursive description of the relevant class of program terms that can be captured in a finite set of program clauses. In particular, consider the following, assuming that symbols beginning with uppercase letters denote instantiatable variables:

1. A program is tail recursive if it contains no recursive calls and its representation can be recognized by the term *(fix λf (abs λm (abs λn (H m n))))*.
2. A program that consists solely of a recursive call with possibly modified arguments is also tail-recursive and its representation must match with the term *(fix λf (abs λm (abs λn (app (app f (E1 m n)) (E2 m n)))))*.
3. Finally, a program is tail-recursive if its body consists of a conditional in which there is no recursive call in the test and whose left and right branches themselves satisfy the requirements of tail-recursiveness. The representation of only such a program unifies with the term

   *(fix λf (abs λm (abs λn (cond (C m n) (T f m n) (E f m n)))))*

   and in a way such that, under the instantiations determined for *T* and *E*,

   *(fix λf (abs λm (abs λn (T f m n))))* and *(fix λf (abs λm (abs λn (E f m n))))*

   represent tail-recursive programs.

These observations provide a complete characterization of the criterion for tail recursiveness under consideration, and they translate immediately into the following λProlog program:

```
(tailrec (fix λf (abs λm (abs λn (H m n))))).
(tailrec (fix λf (abs λm (abs λn (app (app f (E1 m n)) (E2 m n)))))).
(tailrec (fix λf (abs λm (abs λn (cond (C m n) (T f m n) (E f m n)))))) :-
    (tailrec (fix λf (abs λm (abs λn (T f m n))))),
    (tailrec (fix λf (abs λm (abs λn (E f m n))))).
```

Given a program term *Prog*, we can determine whether or not this represents a tail recursive program through a query of the form

```
?- tailrec Prog.
```

Higher-order unification will play an important computational role in this recognition task. In particular, this operation will determine which of the terms in the heads of the clauses matches an incoming program term and, in the case of the last clause, will also aid in destructuring and subsequently constructing terms needed in the recursive calls.

## 2.3 Recursion over Binding Structure

The program for recognizing tail recursiveness just discussed has an obvious defect: it is applicable only to *binary* recursive functions. A question to ask is if we can write a program to carry out such a recognition over *arbitrary* arity functions. Templates are not going to be very useful in this task since these must already anticipate the number of arguments for the function in their structure. A general solution to the problem must instead embody a systematic method for descending under the abstraction corresponding to each function argument; this method can then be applied as many times as is needed in any particular instance before conducting an analysis over the function body.

The scoping primitives present in λProlog provide a means for realizing the necessary recursion over binding structure. The overall computation may, in fact, be structured as follows: The expression that has to be dealt with at the outset has the form *(fix* $\lambda f F$*)*. The objective in this case is to ensure that the free occurrences of $f$ in $F$ are all of a properly restricted kind. Such a check can be carried out by introducing a new constant $c$, annotating this constant so that it can be easily identified later, replacing all free occurrences of $f$ in $F$ with $c$ and analyzing the resulting structure. A *generic* goal can be used to introduce the needed constant, its annotation can be realized by using an *augment* goal to make a special predicate true of this constant and substitution can be realized by application. At the next step, the expression encountered is of the form *(abs* $\lambda x B$*)*. The objective now is to analyze $B$, noting that $x$ may appear freely within this structure. Towards this end, a new constant may be temporarily added to the signature of the object language, the free occurrences of $x$ in $B$ may be replaced with this constant and the resulting term may be further examined. These computations can, once again, be realized using a *generic* and an *augment* goal and function application. After a few repetitions of this step, the 'body' of the function would be reached. This is essentially a first-order structure the needed recursive analysis over which can be specified through Horn clauses.

Assuming a definition for the predicate *term* that allows it to recognize terms corresponding to programs in the object language, the ideas just described translate into the following λProlog program:

```
(tailrec (fix M)) :- ∀f((recfn f) => (trfn (M f))).
(trfn (abs R)) :- ∀x((term x) => (trfn (R x))).
(trfn B) :- (trbody B).
(trbody (cond C M N)) :- (term C), (trbody M), (trbody N).
```

```
(trbody M) :- (recfn M).
(trbody (app M N)) :- (trbody M), (term N).
```

The computation resulting from a use of the above clauses actually mimics the way the recognition task would have been structured had a conventional abstract syntax representation been used. In such a case, it would still be necessary to traverse the body of the function definition, checking the apparent uses of recursion. The advantage with the higher-order abstract syntax style of programming is that tedious bookkeeping steps—such as recording the function name and identifying its free occurrences—receive a simple and logically precise treatment. The practical use of this approach depends, of course, on how efficiently the features supporting it can be realized. In the computation being considered, for example, several substitutions are performed over the function body by forming and contracting beta redexes. Carrying out these substitutions eagerly will result in several walks over the body of the function. Acceptable efficiency is dependent on mechanisms for delaying these substitutions so that they can performed in the same walk that analyzes the function body.

The ideas that we have discussed in this section are quite general in their applicability and they have, amongst other things, been used in encoding computations that arise in theorem proving and manipulation of proofs [2, 6], type checking [13] and the specification of programming language semantics [7]. Moreover, the features that support these ideas have also been widely exploited relative to the metalanguage Elf [30] and its successor Twelf [32]. Thus, the programming benefits of these features seem to be significant, making questions of their implementability important ones to address.

## 3 Implementation Issues and Their Resolution

The computational model underlying λProlog shares many features with the one used for Prolog: the goal to be solved at intermediate stages typically consists of a sequence of simpler ones, there may be choices in clauses to use in solving atomic goals and unification provides the basis for matching an atomic goal with the head of a clause. Logic programming implementations embody mechanisms for dealing with all these aspects: sequential goal structure is realized with structure sharing using environment records and pointers to continuation code, a stack of choice point records is used to succinctly record alternative paths that may be followed on backtracking, the static information present in clause heads is used to compile significant parts of the unification computation and an understanding of how data may become redundant is used to manage the allocation and reclamation of space. Much of this methodology is applicable to λProlog as well. However, there are differences in detail. Considering only the features of the language presently of interest, terms with a richer structure have to be represented, a more sophisticated unification computation has to be realized and different signatures and programs may be relevant to the solution of distinct atomic goals. We discuss the new concerns that arise from these aspects below and outline approaches to their proper treatment within the broader framework.

## 3.1 Representation of Lambda Terms

The usual requirement of a representation for lambda terms is that this support the operation of $\beta$-reduction efficiently. Our special use of these as data structures raises additional concerns. Since it may be necessary to compare or destructure terms during execution, their intensions must be easy to access at run-time. At a logical level, two terms are considered to be equal if they differ only in the names of bound variables. The underlying representation must, therefore, support the rapid determination of $\alpha$-convertibility. With respect to $\beta$-reduction, it is desirable to be able to perform substitutions arising from this operation lazily and also to be able to combine several such substitutions so that they can be performed in the same walk over term structure. Functional programming language implementations embody a simple solution to this problem, and also to questions of $\alpha$-convertibility, but one that gives up an ability that is important in our context: that of examining structure within abstraction contexts.

Explicit substitution notations for lambda calculi that build on the de Bruijn method for eliminating bound variable names provide the conceptual basis for an adequate treatment of these representational questions. A popular version of such a notation is the $\lambda\sigma$-calculus [1]. Our implementation of $\lambda$Prolog uses a different version called the *suspension notation* [21, 28] that we believe is better suited to actual implementation. There are three categories of expressions in this notation that are referred to as terms, environments and environment terms and are given by the following syntax rules:

$$\begin{array}{lll}\langle Term\rangle & ::= & \langle Cons\rangle \mid \langle Var\rangle \mid \#\langle Index\rangle \mid (\langle Term\rangle\ \langle Term\rangle) \mid \\ & & (\lambda\langle Term\rangle) \mid [\![\langle Term\rangle, \langle Nat\rangle, \langle Nat\rangle, \langle Env\rangle]\!] \\ \langle Env\rangle & ::= & nil \mid \langle ETerm\rangle :: \langle Env\rangle \\ \langle ETerm\rangle & ::= & @\langle Nat\rangle \mid (\langle Term\rangle, \langle Nat\rangle)\end{array}$$

In these rules, $\langle Cons\rangle$ and $\langle Var\rangle$ represent constructors and instantiatable variables, $\langle Index\rangle$ is the category of positive numbers and $\langle Nat\rangle$ is the category of natural numbers. Terms correspond to lambda terms. In keeping with the de Bruijn scheme, $\#i$ corresponds to a variable bound by the $i$th abstraction looking back from the occurrence. The expression $[\![t, ol, nl, e]\!]$, referred to as a *suspension*, constitutes a new form of terms that encodes a term with a 'suspended' substitution: intuitively, this corresponds to the term $t$ whose first $ol$ variables have to be substituted for in the way determined by $e$ and whose remaining bound variables have to be renumbered to reflect the fact that $t$ used to appear within $ol$ abstractions but now appears within $nl$ of them. Nominally, the elements of an environment either indicate the retention of an abstraction or are terms generated by a contraction. However, to encode the renumbering of indices needed during substitution, these are annotated by a relevant abstraction level.

In addition to the syntactic expressions, the suspension notation includes a collection of rewrite rule schemata whose purpose is to simulate $\beta$-reduction. These schemata are presented in Figure 1. Of these, the ones labelled $(\beta_s)$ and $(\beta_s')$ generate the substitutions corresponding to the $\beta$-contraction rule on de

| | |
|---|---|
| $(\beta_s)$ | $((\lambda t_1)\ t_2) \rightarrow [\![t_1, 1, 0, (t_2, 0) :: nil]\!]$ |
| $(\beta'_s)$ | $((\lambda [\![t_1, ol+1, nl+1, @nl :: e]\!])\ t_2) \rightarrow [\![t_1, ol+1, nl, (t_2, nl) :: e]\!]$ |
| (r1) | $[\![c, ol, nl, e]\!] \rightarrow c$, provided $c$ is a constant. |
| (r2) | $[\![x, ol, nl, e]\!] \rightarrow x$, provided $x$ is a free variable. |
| (r3) | $[\![\#i, 0, nl, nil]\!] \rightarrow \#(i+nl)$. |
| (r4) | $[\![\#1, ol, nl, @l :: e]\!] \rightarrow \#(nl-l)$. |
| (r5) | $[\![\#1, ol, nl, (t, l) :: e]\!] \rightarrow [\![t, 0, nl-l, nil]\!]$. |
| (r6) | $[\![\#i, ol, nl, et :: e]\!] \rightarrow [\![\#(i-1), ol-1, nl, e]\!]$, provided $i > 1$. |
| (r7) | $[\![(t_1\ t_2), ol, nl, e]\!] \rightarrow ([\![t_1, ol, nl, e]\!]\ [\![t_2, ol, nl, e]\!])$. |
| (r8) | $[\![(\lambda t), ol, nl, e]\!] \rightarrow (\lambda [\![t, ol+1, nl+1, @nl :: e]\!])$. |

**Fig. 1.** Rule schemata for rewriting terms in the suspension notation

Bruijn terms and the rules (r1)-(r8), referred to as the *reading rules*, serve to actually carry out these substitutions. The $(\beta'_s)$ schema has a special place in the calculus: it is the only one that makes possible the combination of substitutions arising from different $\beta$-contractions.

Unification and other comparison operations on terms require these to be first reduced to *head-normal forms*, i.e. to terms that have the structure

$$(\lambda \ldots (\lambda (\ldots (h\ t_1)\ \ldots\ t_m)) \ldots)$$

where $h$, called the head of the term, is a constant, a de Bruijn index or an instantiatable variable. By exploiting the atomic nature of the rules in Figure 1, it is possible to describe a stack based procedure to realize reduction to such a form [20] and to embed this procedure and its use naturally into the structure of a logic programming implementation [23]. Furthermore, the rewriting order can be arranged so as to exploit the $(\beta'_s)$ schema to combine all the substitutions that need to be performed on a term into a single environment. Following this course has measurable advantages: in preliminary studies we have observed benefits in time from following this course as opposed to never using the $(\beta'_s)$ schema that range from 25% to 500% over the entire computation.

We mention a few other salient issues relating to term representation. One of these relates to the internal encoding of applications. With reference to the head-normal form just displayed, there is a choice between representing the subterm under the abstractions as $m$ iterations of applications as the curried presentation indicates or as one application with $m$ arguments. There are practical advantages to the latter: access to the head, with which most structure examination begins, is immediate, the arguments, over which operations have to typically be iterated, are available as a vector and a close parallel to the WAM representation can be used for first-order terms. Our implementation of $\lambda$Prolog therefore uses this representation. In the realization of reduction, there is a choice between destructive, graph-based, rewriting and a copying based one. The former seems to be conservative in both space and time and can be realized using a value trailing mechanism within a WAM-like scheme. Finally, the terms in the suspension notation can be annotated in a way that indicates whether or not substitutions generated by contracting external $\beta$-redexes can affect them. These annotations can permit reading steps to be carried out trivially in certain cases, leading also to a conservation of space and, possibly, a greater sharing in graph-based reduction. Benefits such as these have been noted to be significant in practice [3], a fact confirmed also by our early studies using the *Teyjus* system.

### 3.2 Supporting Higher-Order Unification

The framework for organizing the unification computation relating to lambda terms is given by a procedure due to Huet [8].[3] This procedure considers a set of pairs of terms of the same type with the objective of making the two terms in each pair identical; the set is called a disagreement set and each pair in it is a disagreement pair. Progress towards a solution is made by the repeated application of two different kinds of steps, both of which are based on the normal forms of the terms in a chosen disagreement pair. Referring to a term as flexible if the head of its head-normal form is an instantiatable variable and rigid otherwise, the first kind of step is one that simplifies a pair of rigid-rigid terms. In particular, if the normal forms of these terms are

$$(\lambda \ldots (\lambda (\ldots (h_1\ t_1)\ \ldots\ t_m)) \ldots) \text{ and } (\lambda \ldots (\lambda (\ldots (h_2\ s_1)\ \ldots\ s_n)) \ldots),$$

where the abstractions at the front have been arranged to be of equal length possibly by using the $\eta$-conversion rule, the simplification step concludes that no unifiers exist if $h_1$ and $h_2$ are distinct and replaces the pair with the pairs $\langle t_1, s_1 \rangle$,..., $\langle t_m, s_m \rangle$ otherwise; we note that if $h_1$ and $h_2$ are identical, typing constraints that we have left implicit up to this point dictate that $n = m$. The second kind of step deals with a flexible-rigid disagreement pair and it posits a finite set of substitutions for the variable at the head of the normal form of the flexible term that might be tried towards unifying the two terms. All substitutions must be tried for completeness, leading to a unification process that in general has a branching character.

A useful special case of the application of the simplification step is the one where there are no abstractions at the front of the normal forms of the terms in the disagreement pair. In this situation, this step is similar to the term simplification carried out in first-order unification. Further, when one of the terms is known ahead of time, the repeated application of the step to the pair and, subsequently, to the pairs of subterms it produces can actually be compiled. Finally, if the instantiatable variables within terms appear not as the heads of applications but, rather, as leaves, the simplification process either indicates non-unifiability or produces a disagreement set in which at least one element or each pair is a variable. A generalization of the occurs-check procedure of first-order unification

[3] Huet's original procedure pertains to unifying the *simply typed* lambda terms that are used in $\lambda$Prolog. However, its structure has been utilized for unifying lambda terms with other typing regimens as well.

usually succeeds in the latter case in determining non-unifiability or in finding a most general unifier. The empirical study in [14] concludes that a large percentage of the terms encountered in a $\lambda$Prolog-like language fit the 'first-order' description just considered. Unification of these terms can be treated efficiently and deterministically and it is important to reflect this into an implementation.

There are other situations in which determinism in unification can be recognized and exploited and many of these are embedded in the *Teyjus* system. However, a full treatment of higher-order unification has to eventually contend with branching and structures are needed to realize a depth-first, backtracking based search. The information that is needed for trying a different substitution for a flexible-rigid disagreement pair at a later time can be divided into two parts. One part corresponds to resetting program state to a point prior to making a choice in substitution; in a WAM-oriented model, this includes the values in the argument registers, the program pointer and the continuation pointer. The other part contains information for generating as yet untried substitutions and, if these are unsuccessful, for finding earlier backtracking possibilities. The computation model that is used in $\lambda$Prolog attempts to solve unification problems as completely as is possible before returning to goal simplification. In light of this, the state component of backtracking information will likely be common to more than one unification branch point. A representation and processing scheme can be designed that takes advantage of this situation to create only one record of the state information that is subsequently shared between all the relevant branch point records.

The disagreement set representing a unification problem consists in certain situations of only flexible-flexible pairs. While solutions can be proposed for such problems, this cannot be done without significant redundancy. The most prudent course, therefore, is to treat such sets as constraints on the solution process that are to be resolved when they get more refined. An explicit representation of disagreement sets is necessary for realizing this strategy. The following factors affect the design of a satisfactory representation for this purpose:

1. Disagreement sets change incrementally: they change when substitutions are made for variables, but these typically affect only a few pairs. For this reason, in representing a newly generated set it would be best if unchanged portions of the old set were reused.
2. Backtracking may require a return to a disagreement set that was in existence at some earlier computation point. For efficiency reasons, it should be possible to achieve such a reinstatement rapidly.

Both requirements can be met by using a heap-based doubly linked list representation for the set. The removal of a pair from this list can be realized by modifying pointers in the elements that appear before and after it. For backtracking purposes, it suffices to trail a pointer to the pair. To minimize bookkeeping, the addition of new pairs as a result of simplification must be done conservatively. Structures that may be used to support this are described in [23].

### 3.3 Realizing Generic Goals

The treatment of generic goals on the surface appears to be quite simple: whenever such a goal is encountered, we simply generate a new constant, instantiate the goal with this and solve the resulting goal using the usual logic programming mechanisms. The problem, however, is that scope restrictions have also to be honored. To understand the specific issues, consider the situation in which the program consists of the single clause $\forall x(p\ x\ x)$ and the desire is to solve the goal $\exists y \forall z(p\ y\ z)$. Using the usual treatment of existential goals in logic programming and the suggested one for generic goals, the given goal reduces to one of the form $(p\ Y\ c)$ where $c$ is a new constant and $Y$ is an instantiatable variable. The attempt to solve this atomic goal must now take recourse to matching it with the program clause. If the usual notion of unification is used for this purpose, success will result with $Y$ being bound to $c$. The problem is that this success is not legitimate since the binding for $Y$ allows the new constant to escape out of its scope. Unification must therefore be constrained to prevent such solutions.

The usual logical method for capturing restrictions arising out of quantifier scope is to instantiate universal quantifiers not with new constants but with Skolem functions of the existential quantifiers whose scope they appear within; occurs-check in unification then prevents illegal bindings of the sort just discussed. There is a dual to Skolemization called raising [16] that applies to higher-order logics and this has, in fact, been used in some $\lambda$Prolog implementations. Unfortunately, the logic underlying $\lambda$Prolog is such that the needed Skolem functions (and their duals) cannot be statically determined [19, 34], requiring lists of existential variables to be carried around and used in constructing the needed functions at runtime. Such a scheme is difficult to implement at a low level and certainly cannot be embedded well in an abstract machine.

There is, however, an alternative way to view the instantiation restrictions [19] that does admit an elegant implementation. The central idea is to think of the term universe as being constructed in levels, each new level being determined by a generic goal that is dynamically encountered. Thus, suppose we are given the goal $\exists x \forall y(p\ (f\ x)\ a\ y)$ and that our signature initially consists of only the constructors $f$ and $a$. The collection of terms at the first level consists of only those that can be built using $f$ and $a$. Processing the existential quantifier in the given goal reduces it one of the form $\forall y(p\ (f\ X)\ a\ y)$, where $X$ is a new instantiatable variable. Treating the generic goal now introduces a new constant $c$ and transforms the goal to be solved into $(p\ (f\ X)\ a\ c)$. However, $c$ and all the terms that contain it belong to the second level in the term universe. Furthermore, $X$ can only be instantiated with terms belonging to the first level. A way to encode these constraints is to label constants and variables with level numbers. For a constant, this denotes the stage at which it enters the term universe. For a variable, the label determines the level by which a term must be in the universe for it to constitute a legal instantiation.

An implementation of this scheme amounts to the following. A designated universe level is associated with each computation point and is maintained in a special register. A generic goal increments this register on entry and decrements it on exit; special instructions can realize these effects in a compilation model. Constants and variables that result from generic and existential goals are labelled with the existing universe level at their point of creation. These labels are used whenever a variable needs to be bound during unification. At this time, an occurs-check process ensures that the binding is consummated only if no constant in the instantiation term has a higher label value than that of the variable. The same process also modifies the label on variables in the term to have a value no higher than that on the one being instantiated. This is needed to ensure that refinements to the instantiation term also respect the relevant constraints.

### 3.4 Realizing Augment Goals

Augment goals have the effect of parameterizing the solution of each goal in a sequence by a program. In a sequential implementation, this parameterization can be realized by modifying a central program as computation proceeds. Note that the program may have to change not only as a result of commencing or completing an augment goal, but also as a result of backtracking. From the perspective of efficient implementation, the following are important: Programs should have succinct descriptions that can be recorded easily in choice points. The addition of code upon entering an augment goal should be rapid as also the removal of code. Access to procedure definitions operative at any point in computation should be fast. Finally, compilation of predicate definitions that appear in augment goals should be possible.

We outline here the scheme developed in [22] for realizing all these requirements. The essence of this scheme is in viewing a program as a composite of compiled code and a layered access function to this code, with each augment goal causing a new layer to be added to an existing access function. Thus, consider a goal of the form $(C_1 \wedge \ldots \wedge C_n) \supset G$ where, for $1 \leq i \leq n$, $C_i$ is a program clause with no free variables; this is not the most general situation that needs to be treated, and we will discuss the fully general case shortly. This goal requires the clauses $C_1, \ldots, C_n$ to be added to (the front of) the program before an attempt is made to solve $G$. Now, these clauses can be treated as an independent program fragment and compiled as such. Let us suppose that the clauses define the predicates $p_1, \ldots, p_r$. Compilation then produces a segment of code with $r$ entry points, each indexed by the name of a predicate. In addition, we require compilation to generate a procedure that we call *find_code* that performs the following function: given a predicate name, it returns the appropriate entry point in the code segment if the name is one of $p_1, \ldots, p_r$ and fails otherwise.[4] The execution of the augment goal results in a new access function that behaves as follows. Given a predicate name, *find_code* is invoked with it. If this function succeeds, then the code location that it produces is the desired result. Otherwise the code location is determined by using the access function in existence earlier.

[4] In the *Teyjus* implementation, *find_code* amounts to pointers to a generic search function and a hash-table or a binary search tree.

The described method for enhancing program context is incomplete in one respect: rather than constituting completely new definitions, the clauses provided for $p_1, \ldots, p_r$ may be adding to existing definitions for some of these predicates. Given this, compilation must produce code for each of these predicates that, instead of failing eventually, looks for code for the predicate using the access function existing earlier. Assuming that the last lookup occurs more than once, its effect may be precomputed. In particular, we may associate a vector of size $r$ with the augment goal, the $i$th entry of which corresponds to the predicate $p_i$. One of the actions to be performed on entering the augment goal is then that of using the existing access function with each of the predicates to place pointers to the relevant entry points into this vector.

In a WAM-like implementation, the layered access function can be realized through a structure called an *implication point record* that is allocated on the local stack and that stores the following items:

1. a pointer to an enclosing implication point record representing the previous access function,
2. the *find_code* procedure for the antecedent of the augment goal,
3. a positive integer $r$ indicating the number of predicates defined by the program clauses in the antecedent, and
4. a vector of size $r$ that indicates the next clause to try for each of the predicates defined in the antecedent of the augment goal.

The program context at any stage is completely characterized by a pointer to an implication point record that may be stored in a specially designated *program register*. The goal $(C_1 \wedge \ldots \wedge C_n) \supset G$ may be compiled into code of the form

```
push_impl_point t
   { Compiled code for G }
pop_impl_point
```

where *push_impl_point* and *pop_impl_point* are special instructions that realize the beginning and ending actions of an augment goal and $t$ is the address of a statically created table that stores the *find_code* function for this augment goal and also the numbers and names of the predicates defined. The *push_impl_point* instruction uses its table parameter and the program register to create a new implication point record in an entirely obvious way. The *pop_impl_point* instruction restores the old program context by using the pointer to the enclosing implication point record stored in the one currently pointed to by the program register.

The scheme that we have discussed provides for a particularly simple realization of backtracking behavior as it concerns program context. Since this is given at any point by the contents of the program register, saving these contents in a choice point record at the time of its creation and retaining implication point records embedded under choice points ensures the availability of all the information that is needed for context switching.

The structure we have assumed for augment goals up to this point embodies a simplification. In the most general case, a goal of the form $(C_1 \wedge \ldots \wedge C_n) \supset G$

may appear within the scope of universal quantifiers and the $C_i$s may contain free, non-local, variables. Non-local variables can be treated by viewing a program clause as a combination of compiled code and a binding environment—in effect, as a *closure*—in the scheme described and leaving other details unchanged. Universal quantification over procedure names can lead to two different improvements. First, it may be possible to translate calls to such procedures from within $G$ into a transfer of control to a fixed address rather than to one that is determined dynamically by the procedure $find\_code$. Second, the definitions of such procedures within $(C_1 \wedge \ldots \wedge C_n)$ cannot be extended, leading to a determinism that can be exploited in compiling these definitions and obviating entries for such procedures in the next clause vector stored in implication points.

## 4 The Teyjus System

The full λProlog language includes a typing regimen [26], facilities for conventional higher-order programming [24] and a modules notion for structuring large programs [17] in addition to the features considered in this paper. A curious aspect about the types in the language is that they influence computations. They must, therefore, be present during execution and efficient methods are needed for constructing and carrying them around. In work not discussed here, we have addressed the implementation problems that arise from these other features [10, 23, 27] and have incorporated all our ideas into a virtual machine for λProlog.

An implementation of λProlog based on our ideas envisages four software subsystems: a compiler, a loader, an emulator for the virtual machine and a user interface. The function of the compiler is to process any given module of λProlog code, to certify its internal consistency and to ensure that it satisfies a promise determined by an associated signature and, finally, to translate it into a byte code form consisting of a 'header' part relevant to realizing module interactions and a 'body' containing sequences of instructions that can be run on the virtual machine. The purpose of the loader is to read in byte code files for modules, to resolve names and absolute addresses using the information in the header parts of these files and to eventually produce a structure consisting of a block of code together with information for linking this code into a program context when needed. The emulator provides the capability of executing such code after it has been linked. Finally, the user interface allows for a flexibility in the compilation, loading and use of modules in an interactive session.

The *Teyjus* system embodies all the above components and comprises about 50,000 lines of C code. The functionality outlined above is realized in its entirety in a development environment. Also supported is the use of the compiler on the one hand and the loader and emulator on the other in standalone mode. The system architecture actually makes byte code files fully portable. Thus, λProlog modules can be distributed in byte code form, to be executed later using only the loader/emulator. The source code for the system and associated documentation is available from the URL `http://teyjus.cs.umn.edu/`.

## 5 Conclusion

We have discussed the use and implementation of features of $\lambda$Prolog that provide support for a higher-order approach to abstract syntax. We believe that there are many promising applications for these features, an observation that is accentuated by the extensive use that has been made recently of $\lambda$Prolog and the related language Twelf in prototyping experiments related to the proof-carrying code paradigm. The research we have described here also encompasses issues that are of broad interest to logic programming: matters related to typing, modular development of code and richer notions of equality between terms are important to other interpretations of this paradigm as well and the implementation ideas that we have developed should find applicability in these contexts.

Considerable work remains to be done relative to the *Teyjus* system. Many design choices have been made with little empirical guidance in this first implementation and it is important now to understand their practical implications and to refine them as needed. One category of choices concerns the representation of lambda terms. Explicit substitution notations provide only a framework for this and an actual realization has to address many additional issues such as sharing and optimality in reduction [11, 12], the extent of laziness and destructive versus non-destructive realization. Another possibility not discussed at all here is that of lifting higher-order unification directly to such a notation [5]. Doing this simplifies the construction and application of substitutions but also necessitates bookkeeping steps that may be difficult to realize well in a virtual machine. The *Teyjus* system provides a means to study the pragmatic impact of such choices, a matter often downplayed in theoretical studies related to the lambda calculus. Along another direction, it appears important to improve on the module system and its realization so that it offers stronger support for separate compilation, permits dynamic linking and can be used as a basis for interfacing with foreign code. Another issue relates to garbage collection. The memory management built into a WAM-like implementation has its limitations and these are especially exposed by our use of the heap in carrying out $\beta$-reductions in the *Teyjus* system. An auxiliary system therefore needs to be designed to reclaim disused space. A final question concerns the kind of support to provide for higher-order unification. The present realization of this operation uses a branching search but it may be possible to finesse this, using the ideas in [15] as is done in the Twelf system. Following this course has the additional advantage that the use of types can be limited to compile-time checking, leading to a significant simplification of the virtual machine structure.

We are addressing these and other related questions in ongoing research. We also mention here that $\lambda$Prolog has received other implementations, one of these being the *Prolog/Mali* system [4, 33]. Preliminary experiments with this system indicate a complementary behavior to that of *Teyjus*, with the former being more adept in the treatment of first-order computations and the latter with that of higher-order abstract syntax. This matter needs to be understood better and, where possible, the ideas in *Prolog/Mali* need to be exploited towards enhanced overall performance.

## 6 Acknowledgements

B. Jayaraman, K. Kwon and D.S. Wilson assisted in the design of the λProlog abstract machine and L. Headley, D.J. Mitchell, S.-M. Perng and G. Tong helped in the implementation of the *Teyjus* system. Support for this work has been provided by NSF at different stages under the grants CCR-8905825, CCR-9208465, CCR-9596119, CCR-9803849 and CCR-0096322.

## References

1. M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, 1991.
2. A. W. Appel and A. P. Felty. Lightweight lemmas in λProlog. In *International Conference on Logic Programming*, pages 411–425. MIT Press, November 1999.
3. P. Brisset and O. Ridoux. Naive reverse can be linear. In K. Furukawa, editor, *Eighth International Logic Programming Conference*, pages 857–870. MIT Press, June 1991.
4. P. Brisset and O. Ridoux. The compilation of λProlog and its execution with MALI. Publication Interne 687, IRISA, 1992.
5. G. Dowek, T. Hardin, and C. Kirchner. Higher-order unification via explicit substitutions. *Information and Computation*, 157:183–235, 2000.
6. A. Felty. Implementing tactics and tacticals in a higher-order logic programming language. *Journal of Automated Reasoning*, 11(1):43–81, August 1993.
7. J. Hannan and D. Miller. From operational semantics to abstract machines. *Mathematical Structures in Computer Science*, 2(4):415–459, 1992.
8. G. Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1:27–57, 1975.
9. G. Huet and B. Lang. Proving and applying program transformations expressed with second-order patterns. *Acta Informatica*, 11:31–55, 1978.
10. K. Kwon, G. Nadathur, and D.S. Wilson. Implementing polymorphic typing in a logic programming language. *Computer Languages*, 20(1):25–42, 1994.
11. J. Lamping. An algorithm for optimal lambda calculus reduction. In *Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 16–30. ACM Press, 1990.
12. J.-J. Lévy. Optimal reductions in the lambda-calculus. In J.P. Seldin and J.R. Hindley, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 159–191. Academic Press, 1980.
13. C. Liang. Let-polymorphism and eager type schemes. In *TAPSOFT '97: Theory and Practice of Software Development*, pages 490–501. Springer Verlag LNCS Vol. 1214, 1997.
14. S. Michaylov and F. Pfenning. An empirical study of the runtime behavior of higher-order logic programs. In D. Miller, editor, *Proceedings of the Workshop on the λProlog Programming Language*, pages 257–271, July 1992. Available as University of Pennsylvania Technical Report MS-CIS-92-86.
15. D. Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1(4):497–536, 1991.
16. D. Miller. Unification under a mixed prefix. *Journal of Symbolic Computation*, 14:321–358, 1992.

17. D. Miller. A proposal for modules in λProlog. In R. Dyckhoff, editor, *Proceedings of the 1993 Workshop on Extensions to Logic Programming*, pages 206–221. Springer-Verlag, 1994. Volume 798 of Lecture Notes in Computer Science.
18. D. Miller, G. Nadathur, F. Pfenning, and A. Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.
19. G. Nadathur. A proof procedure for the logic of hereditary Harrop formulas. *Journal of Automated Reasoning*, 11(1):115–145, August 1993.
20. G. Nadathur. An explicit substitution notation in a λProlog implementation. Technical Report TR-98-01, Department of Computer Science, University of Chicago, January 1998.
21. G. Nadathur. A fine-grained notation for lambda terms and its use in intensional operations. *Journal of Functional and Logic Programming*, 1999(2), March 1999.
22. G. Nadathur, B. Jayaraman, and K. Kwon. Scoping constructs in logic programming: Implementation problems and their solution. *Journal of Logic Programming*, 25(2):119–161, November 1995.
23. G. Nadathur, B. Jayaraman, and D.S. Wilson. Implementation considerations for higher-order features in logic programming. Technical Report CS-1993-16, Department of Computer Science, Duke University, June 1993.
24. G. Nadathur and D. Miller. Higher-order logic programming. In D. Gabbay, C. Hogger, and A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 5, pages 499–590. Oxford University Press, 1998.
25. G. Nadathur and D.J. Mitchell. System description: Teyjus—a compiler and abstract machine based implementation of λProlog. In H. Ganzinger, editor, *Automated Deduction–CADE-16*, number 1632 in Lecture Notes in Artificial Intelligence, pages 287–291. Springer-Verlag, July 1999.
26. G. Nadathur and F. Pfenning. The type system of a higher-order logic programming language. In F. Pfenning, editor, *Types in Logic Programming*, pages 245–283. MIT Press, 1992.
27. G. Nadathur and G. Tong. Realizing modularity in λProlog. *Journal of Functional and Logic Programming*, 1999(9), April 1999.
28. G. Nadathur and D. S. Wilson. A notation for lambda terms: A generalization of environments. *Theoretical Computer Science*, 198(1-2):49–98, 1998.
29. G.C. Necula. Proof-carrying code. In *24th Annual ACM Symposium on Principles of Programming Languages*, pages 106–119. ACM Press, January 1997.
30. F. Pfenning. Logic programming in the LF logical framework. In G. Huet and G.D. Plotkin, editors, *Logical Frameworks*, pages 149–181. Cambridge University Press, 1991.
31. F. Pfenning and C. 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.
32. F. Pfenning and C. 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.
33. O. Ridoux. MALIv06: Tutorial and reference manual. Publication Interne 611, IRISA, 1991.
34. N. Shankar. Proof search in the intuitionistic sequent calculus. In D. Kapur, editor, *Automated Deduction – CADE-11*, number 607 in Lecture Notes in Computer Science, pages 522–536. Springer Verlag, June 1992.
35. D.H.D. Warren. An abstract Prolog instruction set. Technical Note 309, SRI International, October 1983.