The Architecture of an Implementation of LambdaProlog: Prolog/Mali

Pascal Brisset, Olivier Ridoux

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

The Architecture of an Implementation of LambdaProlog: Prolog/Mali

Pascal BRISSET et Olivier RIDOUX

N° 2392

Octobre 1994

PROGRAMME 2

ISSN 0249-6399

The Architecture of an Implementation of LambdaProlog: Prolog/Mali

Pascal BRISSET* et Olivier RIDOUX**

Programme 2 — Calcul symbolique, programmation et génie logiciel
Projet Lande

Rapport de recherche n°2392 — Octobre 1994 — 19 pages

Abstract: λProlog is a logic programming language accepting a more general clause form than standard Prolog (namely hereditary Harrop formulas instead of Horn formulas) and using simply typed λ-terms as a term domain instead of first order terms. Despite these extensions, it is still amenable to goal-directed proofs and can still be given procedural semantics. However, the execution of λProlog programs requires several departures from the standard resolution scheme. First, the augmented clause form causes the program (a set of clauses) and the signature (a set of constants) to be changeable, but in a very disciplined way. Second, the new term domain introduces the need for a β-reduction operation at run-time, and has a semi-decidable and infinitary unification theory. MALI is an abstract memory designed for storing the search-state of depth-first search processes. Its main feature is its efficient memory management. We have used an original λProlog-to-(C+MALI) translation: predicates are transformed into functions operating on several continuations. The compilation scheme is sometimes an adaptation of the standard Prolog scheme, but at other times it has to handle new features such as types, β-reduction and delayed unification.

Key-words: Implementation, Logic Programming, λProlog, Memory Management, Mali

(Résumé : tsvp)

*ENAC, 7 av. Édouard Belin, BP 4005, 31055 Toulouse Cedex, pbrisset@eis.enac.dgac.fr
**ridoux@irisa.fr

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : (33) 99 84 71 00 – Télécopie : (33) 99 84 71 71

L'architecture d'une implémentation de LambdaProlog: Prolog/Mali

Résumé : λProlog est un langage de programmation logique qui accepte une forme de clause plus générale qu'en Prolog standard (les *formules héréditaires de Harrop* au lieu des *formules de Horn*) et qui a pour domaine de calcul les *λ-termes simplement typés* au lieu des *termes de premier ordre*. En dépit de ces extensions, λProlog est encore compatible avec un schéma de preuve dirigé par le but, et on peut lui donner une sémantique opérationnelle. Cependant l'exécution des programmes λProlog nécessite quelques additions au schéma standard. Premièrement et à cause de la forme plus générale des clauses, le programme (un ensemble de clauses) et la signature (un ensemble de constantes) peuvent changer pendant l'exécution, mais d'une manière très disciplinée. Deuxièmement et à cause du nouveau domaine de calcul, on doit effectuer des β-réductions lors de l'exécution, et le problème d'unification qui doit être résolu à chaque étape de calcul est semi-décidable et infinitaire. MALI est une mémoire abstraite conçue pour gérer un processus de recherche en profondeur d'abord. Sa caractéristique principale est sa gestion de mémoire efficace. Nous avons utilisé un schéma de traduction original de λProlog vers (C+MALI). Les prédicats sont traduits en des fonctions qui opèrent sur plusieurs continuations. Ce schéma est applicable à Prolog standard, mais il tient aussi compte des nouveautés de λProlog comme les types, la β-réduction et la suspension de l'unification.

Mots-clé : Implémentation, Programmation logique, λProlog, Gestion de mémoire, Mali

1 Introduction

The logic programming language λProlog [30, 29, 31, 17, 15, 28, 16, 32] improves on standard Prolog because it features more powerful operations on terms and programs while still giving them a logical semantics. A keyword common to all these features is *scoping*. λ-Terms introduce scoping at the term level, explicit quantifications (universal and existential) introduce scoping at the formula level, and the deduction rules for explicit quantification and implication introduce scoping in proofs, i.e. at a dynamic level. Deduction rules for λProlog are usually given in the framework of sequent proofs.

λProlog requires some implementation effort for being able to compete with Prolog in efficiency (and then in popularity). Another condition for popularity is to overcome the idea that it is a "difficult" language, but this is another story. The initial implementation of λProlog by Miller and Nadathur, and the second one, eLP, by the Ergo Project at Carnegie-Mellon University, were far from being able to compete with Prolog. Since then, a few teams have worked on the implementation of λProlog. As far as we know, current teams are Nadathur, Kwon and Wilson at Duke University [22, 21, 36], Jayaraman at the University of Buffalo (formerly with Nadathur), Elliott and Pfenning at CMU [14], Felty and Gunter at Bell Labs, and the authors at Inria.

Other works are done in a similar framework for integrating linear logic and logic programming (Pareschi and Andreoli [3], Hodas and Miller [19]), or higher-order type systems and logic programming (Elliot [13], Pfenning [38, 39]).

We present in this paper the broad lines of our implementation of λProlog: Prolog/Mali. We have implemented λProlog for its own merits, and as a demonstration that memory management issues are a good guide for implementing logic programming systems. Speed was always our second concern.

We assume a knowledge of Prolog and λProlog, their semantics, and their basic algorithms: logical variable, search-stack, unification, λ-unification [20], deduction rules, and uniform proofs [34, 32]. We adopt an architectural presentation: in section 2, we present the kernel subsystem that is in charge of the elementary representation problems, in section 3, we present a software layer which is both a specialization and an extension of the kernel, finally, in section 4, we present the compilation scheme. We conclude in section 5.

A preliminary version of this paper was presented at the Workshop on λProlog held in Philadelphia in 1992 [8].

2 MALI

MALI [6, 40] (Mémoire Adaptée aux Langages Indéterministes — memory for non-deterministic languages) can be specified as the abstract data type *stack of mutable first-order terms*. This abstract data type encompasses the representation of the state of every logic programming language that performs a depth-first search in a search-tree. MALI is the name of a general principle that has several implementations. The name of the implementation we used in Prolog/Mali is *MALIv06*.

We present what MALI brings to the overall system, and, to avoid any ambiguity, what it leaves undone.

2.1 What MALI Brings to Prolog/Mali

2.1.1 A Data-Structure

MALI brings an abstract data-type which we call *MALI's term*. MALI's terms may be described more concretely as graphs with nodes that can be reversibly substituted. MALI's terms are organized in a *term-stack* which is itself a term. A collection of node constructors is offered, among them *atoms* (i.e. leaves), *compound nodes* (i.e. cons or tuples), and *levels* (i.e. term-stack constructors). Some of the compound node constructors are called *mutable* constructors, and the terms constructed with them are called *muterms*. Mutable nodes can be subject to *reversible substitutions*, according to a discipline that is close to the substitution of logical variables in Prolog. According to the discipline, muterms, substitutions, and the term-stack are in the same relationship as logical variables, substitutions, and the search-stack of Prolog. For every kind of node constructor, commands and operations exist for creating and reading them, and for accessing their subnodes (if any). Commands also exist for substituting terms for muterms, and for manipulating the term-stack (pushing, popping, and pruning the term-stack).

Every node constructor can be given an elementary typing via the use of *sorts*. This makes it possible to "decompile" the representation of an application term. For instance, Prolog's integers and constants can be both represented by MALI's atoms, which must be discriminated by their sorts.

In the sequel, we note[1]

- *(le S R N)* a term-stack (a *level*, in MALIv06's jargon) of sort S, top value R (a *root*) and substack N,
- *(at S V)* an atom of sort S and value V,
- *([m]c0 S)* a, possibly mutable, nullary compound term of sort S,
- *([m]c2 S T1 T2)* a, possibly mutable, binary compound term of sort S and subterms *T1* and *T2*,
- and *([m]tu S N T1 ... Tn)* a, possibly mutable, compound term of sort S and N subterms *T1* to *Tn*.

We use a labelled notation, *label@term* and *label*, to note different occurrences of the same term. A term may have several labels through substitution, *label1@label2@term*. Terms with labels in common share the same representation; they must be compatible up to a substitution. Terms with different labels (or no label) are different even if they have the same notation; to apply a substitution to one has effect on the others only through occurrences of shared subterms.

It should be clear from this short description that *one of* the intended usages of muterms and the term-stack is the implementation of logical variables and of a search-stack. However, this is the only commitment with logic programming, and other usages are possible. MALI knows nothing about the basic mechanisms of Prolog (resolution, unification), or about λProlog's deduction rules and λ-terms.

2.1.2 A Memory Management

MALI's terms need memory for their representation. It is automatically allocated when creating terms, and deallocated by a garbage collector. Deallocation is optimal with respect to the level of knowledge that is available to MALI. The restriction means that application-dependent accessibility properties are not taken into account by MALI. They can be taken into account indirectly by a proper mapping of the application structures onto MALI's terms.

We call *usefulness logic* of a programming language the relation that describes which run-time data-structures are useful only considering the operational semantics of the language. The usefulness logic of the core of logic programming is that *"Every useful term is accessible from some search-node under the binding environment of the* same *search-node"*.

The usefulness logic of the core of functional programming says that *"Every useful term is accessible from some root";* binding environments are not mentioned.

So, if one uses a functional programming system for implementing a logic programming system and nothing special is done for memory management, the usefulness logic that is actually implemented cannot be more precise than *"Every useful term is accessible from some search-node under some binding environment".* It is usually worse and considers the union of all the binding environments.

The two important features of MALI's memory management are *early reset* and *muterm shunting*. Early reset causes substitutions to be undone[2] by the memory manager seeing that some muterm is never accessible when substituted. Muterm shunting means that substitutions, which are created reversible, may be made definitive[3] seeing that some substituted muterm is never accessible when not substituted. These two features are described at great length in the MALIv06 tutorial [40]. They are also described by other authors [12, 4, 41].

Commands exist for controlling memory management: supplying MALI with memory, taking useless memory from MALI, or starting a garbage collection.

2.1.3 Debugging Tools

MALI offers debugging tools for assisting a user in the development of an application. Debugging tools allow to check preconditions of commands, to display components of MALI's state, and to trace commands.

It is important that at every level of an architecture (software or hardware) debugging tools are available. It makes the complexity of composing layers tractable. We will not dwell too long on this subject in other sections; it is enough to know that the specialized intermediate machine also has debugging tools for checking a fair use of everything it defines. The Prolog/Mali system also has debugging tools, but the ultimate level is the level of λProlog applications which should also come with their debugging tools. This is up to the discipline of λProlog users.

[1] The notation is only a convenience for commenting on MALI's term; it is not part of the programming interface.
[2] Without waiting for backtracking. Hence the name "Early reset".
[3] The effect is to collapse chains of substitutions. Hence the name "Muterm shunting".

2.2 What MALI Leaves Undone

2.2.1 A Memory Policy

We distinguish memory management inside an application, which aims at improving the use of some memory supplies, and the management of memory at the interface with a host system, which aims at configuring the supplies.

We call *memory policy* the set of decisions related to memory supplies. The decisions range from the amount of memory supplied to MALI, the way this memory amount evolves, to the amount of computing power dedicated to memory management ($\approx$ the frequence of the calls to the garbage collector). A memory policy can be very sophisticated because it deals with many interrelated parameters. For instance, it is likely that, in order to diminish the computing power dedicated to memory management, the total memory allocated to MALI must be increased. However, supplying more memory to MALI may decrease the availability of the host system.

Elementary commands for designing a sophisticated memory policy are available in MALI, but no policy is specified.

2.2.2 Application Level Terms and Execution Scheme

The only commitment of MALI with logic programming is the term-stack and the muterm substitution. Everything remains to be done as for the representation of the data-structures of an application. The implementor must find a mapping from its application terms onto MALI's terms. In λProlog for instance, the representation of simply typed λ-terms, their unification and normalization must be mapped on MALI's terms, and on procedures using MALI's commands and operations. In both Prolog and λProlog, the search procedure must be mapped on MALI's terms and commands.

The hint for mapping application terms and their operations is that it is intended that muterms and the term-stack represent logical variables and a search-stack.

MALI offers an efficient memory management but brings no solution to the time efficiency. The packaging of MALIv06 is designed to hinder as little as possible any effort to yield speed efficiency.

2.2.3 Program Representation

MALI has no notion of program. It is not even intended that an application level program should be represented in MALI. This is a totally independent issue.

3 A Specialized Intermediate Machine

We have designed a specialized intermediate machine (SIM[4]), of the level of the WAM [42, 2], for filling parts of the gap between MALI and λProlog.

The SIM is a specialization of MALI because it forces some interpretation on MALI's terms. It is also an extension of MALI because it defines new notions that have no equivalent in MALI (e.g. unification, continuations). As a specialization of MALI, the SIM defines specialized node constructors, and commands and operations for creating, reading, and traversing them. As an extension, it defines commands for implementing the new notions for every specialized node constructors they apply to.

The SIM still says nothing of what is a program, and how to use the machine efficiently. This is up to the compilation scheme. We review what the SIM brings to the overall system.

3.1 λProlog Terms

To choose a representation for terms in the context of λProlog is a new problem because the requirements of Prolog technology, of simply typed λ-calculus, and of uniform proofs of hereditary Harrop formulas must be met at the same time.

Prolog technology requires the representation of logical variables and substitutions. It also requires that substitutions be reversible because the search for a proof is done by a depth-first traversal of a search-tree.

Simply typed λ-calculus requires the representation of abstraction and application, the representation of types, and the capability to compute at least long head-normal forms because the unification procedure needs them. To meet the first requirements, long head-normalization should be reversible too.

[4] SIM is not a brand name for *this* specialized intermediate machine; it only designates *this* layer in a software architecture using MALI.

Proving hereditary Harrop sequents is required to represent universally quantified variables and to check the correction of signatures. It also requires the handling of implied clauses but this has little to do with our representation of terms.

We only describe our implementation choices. The motivations are discussed in a technical report by the same authors [9], and in the thesis of the first author [7].

3.1.1 Types

One of the differences between Prolog and λProlog is that the terms of λProlog must be typed for λ-unification to be well defined. Huet's procedure deals with simply typed λ-terms, but λProlog extends simple types with type variables (type schemes). This results in *generic polymorphism*.

Follows a sample declaration for polymorphic homogeneous lists and a polymorphic ternary relation on them.

```
kind list type -> type .
type [] (list A) .
type '.' A -> (list A) -> (list A) .
type append (list A) -> (list A) -> (list A) -> o .
```

The list *[1,2]* can be represented in MALIv06 like

```
(c2 S_LIST (at S_INT 1) (c2 S_LIST (at S_INT 2) (c0 S_NIL))) .
```

The type *A->(list A)->(list A)* can be represented in MALIv06 like

```
(c2 S_ARROW A@(mc0 S_UNK_T)
     (c2 S_ARROW listA@(tu S_APPL_T 2 (at S_SYMB_T list) A) listA)) .
```

Type unknown *A* is represented as a mutable nullary compound because it must be reversibly substitutable[5], and it has no other information associated to it. Its representation is shared by all its occurrences (label *A*). Note the sharing of *(list A)* indicated by the use of label *listA*. This sharing is not mandatory; it only diminishes memory consumption.

The idea of generic polymorphism in λProlog (or Typed Prolog [25] as well) is that: *"Types of different occurrences of a constant are independent instances of its type scheme, and types of different occurrences of (any kind of) a variable are equal".*

One must also choose whether a clause of the program can be selected on grounds of the type of its predicate symbol or not. We have chosen to forbid selecting a clause on these grounds. It means we follow the *definitional genericity principle* [25]: *"Types of different body occurrences of a predicate constant are independent instances of its type scheme, whereas types of head occurrences are renamings of the type scheme".*

With this principle, type inference leads to a non-uniform semi-unification problem which has been shown to be undecidable by Kfoury, Tiuryn and Urzyczyn [23]. In our implementation, types of constants (predicative or not) are only checked, and types of (any kind of) variables are inferred.

The reason for sticking to definitional genericity is that it is the most natural when predicates are seen as definitions and type schemes as abstractions of the definitions. It is also required for allowing a simple but sound modular analysis of programs. We want to be able to type-check a module using the type schemes of the modules it imports but not the modules themselves.

In λProlog, it is necessary to represent types at run-time for controlling unification, and some conditions are missing for having a *semantic soundness* result of the kind "Well typed programs cannot go wrong"[6].

The problem with semantics soundness is that nothing restricts λProlog constants to have the *type preserving property* [18]: *Every type variable in a type scheme should appear in the result type (the type to the right of the right-most ->).* The advantage of having the type preserving property is that the types of the subterms of a term built with a type preserving constant can be inferred from the type of the term. The disadvantage is that it is not flexible enough for representing dynamic types [1].

[5] Types must be unified at run-time.

[6] In this context, "going wrong" means "trying to solve ill-typed unification problems".

Types for "not going wrong" We call *forgotten type variables* the type variables that do not occur in the result type of a non-preserving type scheme. We call *forgotten types* the instances of the forgotten type variables. Only forgotten types need to be represented at run-time for avoiding "going wrong". They must be attached as supplementary arguments to the term constructors that are not type preserving. These pseudo-arguments must always be unified before the regular arguments. This makes λ-unification problems always well-typed.

In fact, what is implemented is the representation and unification of terms of a polymorphic type system [5]. It is as if a symbol defined as

kind dummy type .
type forget _ -> dummy .

were defined as

type forget_ 'PI' A\(A -> dummy) .

where *'PI'* is the product type quantifier, and a term like *(forget 1)* were *(forget_ int 1)*. The term *(forget_ int 1)* can be represented in MALIv06 like

(tu S_APPL 3 (at S_SYMB forget_) (at S_SYMB_T int) (at S_INT 1)) .

Unlike Typed Prolog [33, 25], there is no special syntax in λProlog for declaring predicate constants. They are only distinguishable by their result type, *o*. Every predicate constant forgets every type variable in its type because its result type contains no type variable. It can be shown that if the predicates obey the definitional genericity principle, unification of these forgotten types will always succeed; type unification of types forgotten by predicate constants is only required for conveying types along the computation. In a system that does not need that conveying (e.g. standard Prolog), the forgotten types of predicate constants need not be represented [33, 18]. In λProlog, conveying the types is required for controlling unification.

Types for controlling projection in unification Let us first recall the core of Huet's λ-unification procedure [20].

For a pair $<\lambda\tilde{x}(F\ \tilde{s}_p)\ ,\ \lambda\tilde{x}(\Psi\ \tilde{t}_q)>$, where F is a *flexible* head (a logical variable) and Ψ is a *rigid* head (not a logical variable), at most $p+1$ substitutions are produced by two rules:

1. if Ψ is a constant, the *imitation* rule produces $[\ F \leftarrow \lambda\tilde{u}_p(\Psi\ \tilde{E}_q)]$,
2. for each $0 \leq i \leq p$ such that $\tau(s_i) = \tau_1 -> \ldots \tau_m -> \tau((F\ \tilde{s}_p))$, the *projection* rule produces $[\ F \leftarrow \lambda\tilde{u}_p(u_i\ \tilde{E}_m)]$.

Every E_k in $\tilde{E}_q$ or $\tilde{E}_m$ stands for $(H_k\ \tilde{u}_p)$, where H_k is a new logical variable with the appropriate type.

For the precondition of the projection rule (2. above) being testable at run-time it is enough that logical variables are equipped with their types. The types of logical variables themselves need never be unified because when a unification problem is to be solved then it is well-typed (i.e. the two terms of the problem have identical types). This is a side-effect of unifying first the forgotten types in the pseudo-arguments and then the regular arguments.

In λProlog, nothing prevents having a type with a variable result type. This makes the checking of the type condition unsafe: there can be no argument satisfying the condition in some binding environment while projection is possible in a more precise binding environment. The only safe solution is to suspend unification until the result type get known. However, the traditional solution is to commit the result type to be a constant [37]. We believe that nothing satisfactory will be done before these *flexible types* are better understood.

Types for new logical variables It is easy to attach a type to logical variables coming from the program: it is an outcome of the type inference/checking. But the imitation and projection rules of λ-unification introduce new logical variables that correspond to nothing in the source program (the H_k's). A type must be attached to them anyway. They all have types $\nu_1 -> \ldots -> \nu_p -> \Phi$ where the ν_i's are the types of the arguments of the flexible head, and Φ depends on the rule.

In case of projection (see above in the condition controlling projection), the Φ of every new logical variable H_j is τ_j. It is given by the type of variable F. In case of imitation, the Φ is the type of the corresponding argument of the rigid head. So, it remains to be able to infer the types of rigid heads in unification problems.

There are three kinds of rigid heads: λ-variables (but they cannot be imitated), function constants, and universal variables (they are introduced for solving universally quantified goals). First, we attach their type

to every universal variable. Second, we observe that the type scheme of a constant, plus the forgotten types attached to it, plus the result type[7], give enough information for reconstructing the full type of the constant. A type reconstructing function is generated at compile-time from every type scheme declaration as follows.

Let $\mathcal{U}$ be a function returning the most general unifier of two types τ_1 and τ_2. To every constant with type scheme $\Pi\tilde{t}_m(\sigma\ \tilde{t}_m)$ [8], whose result type is ρ [9], we associate the type reconstruction function $\lambda\tilde{t}_m\tau(\mathcal{U}(\rho,\tau)(\sigma\ \tilde{t}_m))$. The type reconstruction function must be applied to the actual forgotten types of some occurrence of the constant and to the result type of the matching unknown. Type checking makes it sure that $\mathcal{U}(\rho,\tau)$ defines a substitution.

Let the following declaration serves as an example.

```
kind t type -> type.
type f A -> B -> (t B).
```

Type variable A is forgotten, hence the declaration is read as

```
type f_ 'PI' A\(A -> B -> (t B)).
```

The type reconstruction function is $\lambda a\tau(\mathcal{U}((\text{t B}),\tau)(a\text{->}B\text{->}(t\ B)))$.

With these declarations, unification problem *<(F 1),(f 1 [1])>* is actually represented as *<(F 1), (f_ int 1 [1])>*, knowing that variable F has type *(int->(t (list int)))*. Imitation yields substitution $[F\leftarrow\lambda x(f\_\ int\ (H_1\ x)\ (H_2\ x))]$ which introduces new unknowns H_1 and H_2. For typing them, the type reconstruction function is called with types *int* and *(t (list int))* as parameters. It unifies *(t (list int))* and *(t B)*, producing substitution $[B\leftarrow(list\ int)]$, and applies the substitution to the type scheme in which the forgotten type variable is replaced by *int*. So, the actual type of this occurrence of constant f is reconstructed: *int->(list int)->(t (list int))*. Thus, unknowns H_1 and H_2 are given types *int->int* and *int->(list int)*.

3.1.2 λProlog Terms

Terms are represented using the full copy technique (as opposed to structure-sharing or a mix of structure-sharing and copy) for memory management reasons: this gives the most precise allocation/deallocation operations for any type of control, and λProlog needs to depart from the standard control.

A novelty of λProlog is that terms need normalization. In Prolog/Mali, normalization alters the representation of term for sharing reduction effort, and also for memory management. λ-terms are represented by graphs, and normalization is implemented as graph-reduction.

Abstractions and applications We will see that logical variables are not the only application level structures that can be represented by MALI's muterms.

Abstractions and applications are represented by reversibly mutable graphs, so that it is possible to physically replace a redex by its reduced form in the graph. This provides sharing of the reduction effort. Reversibly means that mutations (reductions) can be undone when backtracking. This is the result of inserting graph reduction in a Prolog context.

Substituting new representations for older ones in a reversible way forces to store all the history of every term representation. However, MALI's memory management, especially muterm shunting, will remove every useless old representation. Muterm shunting shortens the history of term representations.

Terms are represented as much as possible in their long head-normal form. So, abstractions and applications are in fact tuples of nested elementary abstractions and applications. The term *λnsz(s (n s z))* can be represented in MALIv06 like

```
(mtu S_ABST 4 n@(c0 S_VAR) s@(c0 S_VAR) z@(c0 S_VAR)
      (mtu S_REDEX 2 s (mtu S_REDEX 3 n s z))) .
```

The applications are potential redexes, hence the sort *S_REDEX*. Note the labels *n*, *s*, and *z* for sharing the representations of λ-variables.

[7] In the context of unification, it can be found in the flexible head.
[8] As suggested above, forgotten types are explicitly quantified.
[9] By definition, it cannot depend on forgotten types.

First-order terms We call informally *first-order terms* the rigid terms whose head is a constant, and whose result type is not a variable. They are distinguished as much as possible because they are definitely in long head-normal form and they can be unified using a cheaper procedure. Note that rigid terms whose head is a constant, but whose result type is a variable may need be η-expanded at run-time when the variable result type gets bound to an arrow type.

Universal variables and logical variables In the following, we say that a logical variable *captures* a term if it is bound to a value that contains the term.

Universal variables are among the new constructs of λProlog that enforce checking scoping conditions. A universal variable can be captured by every logical variable of its scope, whereas it cannot be captured outside its scope. I.e. in context $\forall x \exists U \forall y$, universal variable x can be captured by logical variable U, but y cannot. Since λ-variables are essentially universal, they are always bound in the innermost part of the context. So, they can never be captured by logical variables. Constants are also essentially universal, but they are always bound in the outermost part of the context. So, they can always be captured by logical variables.

Scopes of universal variables are represented by their nesting level. A nesting level is attached to every logical variable and every universal variable, and a register contains the value of the current nesting level. When a universally quantified goal is executed, the nesting level register is first incremented, and then a new universal variable is created with the new nesting level value. Every further creation of logical variables within the scope of this goal but out of the scope of any nested universal quantification will be done with the new nesting level value. We assume that the initial nesting level is 0.

Given that logical variables and universal variables must also carry their types, but the former are substitutable while the latter are not, they can be represented in MALIv06 as follows

(mc2 S_UNK type (at S_SIG nesting_level))
(c2 S_UVAR type (at S_SIG nesting_level))

When an attempt is made to substitute a term for a logical variable, the scopes of the term and all its subterms are checked using the nesting levels. If the term contains universal variables of a higher nesting level than the logical variable then the substitution is illegal. If the term contains logical variables of a higher nesting level than the substituted logical variable then their nesting levels should be lowered to the nesting level of the substituted logical variable. If a universal variable or a logical variable with a higher nesting level is in fact in an argument of a flexible term then the scope-checking must be suspended because the problematical universal variable or logical variable may disappear as a side-effect of another substitution. This shows that an implementation of λProlog must use the technology of Constraint Logic Programming. For instance, $[X^1 \leftarrow (U^1\ 1\ Y^2)]$ is a problematical substitution[10], but after substitution $[U^1 \leftarrow \lambda xy(F^1\ x)]$ is applied, it is no more problematical.

We have seen that logical variables cannot capture λ-variable, and can only capture universal variables whose scope they belong to. A flexible term can be seen as a generalization of the logical variable which is explicitly allowed to capture supplementary terms (the arguments). For instance, term X^1 above is a generalized logical variable that is implicitly allowed to capture the universal variable of level 1 and every constant, and is explicitly allowed to capture 1, which is only redundant because it was already implicit, and Y^2, which is not redundant because of the nesting level of U^1.

One of the effects of substituting a term to a logical variable is to diminish the allowance of a generalized logical variable (see the same flexible term after substitution $[U^1 \leftarrow \lambda xy\ (F^1\ x)]$ is applied). Allowance cannot increase because the binding value of a logical variable must be in its allowance. The main consequence is that no decision procedure related to the occurrence of some patterns can be complete when involving flexible terms. We have exposed it for scope-checking, but it is also true for the occurrence-check in unification. If the pattern has an occurrence in a flexible term, a substitution may take it away.

3.1.3 Reduction

The issues raised by reduction in λProlog are somewhat different from those raised in functional programming because reduction must go through abstractions, which is almost never the case in functional programming, β-redexes may appear at run-time by the effect of substitutions that are solutions of unification problems, and the need for reducing comes from the unification procedure, which asks only for η-expanded head-normal forms.

Reduction is implemented as graph-reduction. Since abstractions and applications are represented as tuples, reduction considers several β-redexes at once. This saves term traversing and duplication, hence time and memory.

[10] The nesting levels are written as superscripts.

The basic scheme is to duplicate the left-most part of a redex, and to replace λ-variables occurrences by the arguments. A critical improvement over the basic scheme is to recognize combinators which are subterms of the left-most part of redexes; they need not be duplicated. Every logical variable, every goal argument, and every instance of a term that is a combinator is a combinator. This shows that many terms are combinators and that once a combinator is detected it is safe to tag it as such. Tagging amounts to having more sorts for representing the terms of the cross-product (combinator/non-combinator)$\times$(abstraction/application).

This improvement is fundamental and changes the complexity of useful λProlog predicates [11]. It is not committed to our architecture; it only has to do with reduction. Another improvement is to recognize abstractions that are η-reducible. These abstractions are required in the η-expanded head-normal form, but β-redexes that are formed with them do not need the duplication phase. This improvement also changes the complexity of useful λProlog predicates [9].

3.1.4 λ-Unification

The now conventional names for the different procedures of λ-unification are SIMPL, MATCH and TRIV. We add to them a specialized unification command of the SIM for every kind of term constructors, and a first-order unifier, UNIF1. The main idea is to consider the different unification procedures as as much sieves. If a unification problem cannot be handled by a procedure it is passed to the next one.

Specialized unification commands A sequence of specialized unification commands is generated by the compiler for every clause head. Specialized unification commands can be seen as resulting from a partial evaluation of the general unification procedure. In case there is not enough information in the head (e.g. a second occurrence of a logical variable), the control is passed to procedure UNIF1. This is much like what is done in standard Prolog systems. In case the head term is higher-order, they only build a representation of the unification problem and pass it to SIMPL.

UNIF1 A first-order unification procedure, UNIF1, is used as much as possible on the "first-order terms" (terms with sort S_APPL) until a higher-order term is met.

SIMPL When a higher-order term pops up in UNIF1 or in the specialized unification commands, one switchs to procedure SIMPL. Its outcome is a set of flexible-rigid pairs. If it is not empty it is passed to MATCH, otherwise, a success is reported. Procedure SIMPL may report a failure if a clash of constants or λ-variables occurs.

MATCH Procedure MATCH is the non-deterministic part of λ-unification. It is described as the core of Huet's procedure in section 3.1.1.

Its non-determinism and the one coming from the proof-search are merged in a single search process. To do the merging easily, we write the non-deterministic choice between imitation and projections in λProlog. The actual imitation and projection rules are implemented as deterministic built-in predicates.

Suspensions Flexible-flexible pairs cannot usually be solved as such because they have too many arbitrary solutions. They are suspended. We use the versatility of MALI's muterms for encoding the suspended flexible-flexible pairs within the flexible heads as a constraint. As soon as one of the flexible heads becomes bound, its constraints are checked. This is similar to the *attributed variable* technique described by Le Huitouze [26].

TRIV A flexible-rigid pair is not passed directly to procedure MATCH, nor is a flexible-flexible pair automatically suspended. They are first passed to procedure TRIV, which tries to solve them in a fast deterministic way. TRIV applies various heuristics; if none works the pair is actually passed to MATCH or suspended.

The heuristics aim at finding pairs of the form $< X, t >$ under various disguises. If such a pair is discovered and logical variable X does not occur in term t then $[X \leftarrow t]$ is the solution to the unification problem. In a way similar to the scope-checking in section 3.1.2, the occurrence-check is more complicated than for the first-order case because not all occurrences of X in t are dangerous. If one is found and it is dangerous then unification fails. If it is not dangerous then TRIV passes the pair to MATCH.

A good TRIV procedure must recognize a trivial pair under such disguise as

1. $< \lambda x.(X\ x), t >$, which is η-equivalent to a trivial pair,

2. $< X^i\ u^{i+1} \ldots u^{i+j}, t >$, where the superscripts represent the scope nesting, and the u^k's are universal variables.

The second disguise is equivalent to $< X'^{i+j}, t >$ for a new logical variable X'. In this case, the solution substitution is $[X^i \leftarrow \lambda x_1 \ldots x_j\,([u^{i+1} \leftarrow x_1] \ldots [u^{i+j} \leftarrow x_j]t)]$ for taking into account the disguise. This is a very frequent case because a lot of λProlog programming is about exchanging universal variables and λ-variables (i.e. essentially universal quantification at the formula level and essentially universal quantification at the term level). The following predicate is an example of the exchanging trick:

```
type list2flist (list A) -> ((list A) -> (list A)) -> o .
list2flist L FL :- pi list\(conc L list (FL list)) .
```

The predicate relates the standard representation of a list (say *[1,2,3]*), and its functional representation (*z\[1,2,3|z]*) [11].

Folding representations The logic of unification is to find a substitution making two terms equal. If they are equal then they can share the same representation. We have seen that both abstractions and non-first-order applications are represented by muterms. So, it is easy to make the two terms share the same representation by substituting one for the other. The effect is to fold the representations because two terms with initially different representations end up to have the same. This substitution must be reversible (like the others: solution substitution and λ-reduction substitution). Reversibility comes as a consequence of using muterms. Folding saves unification effort because identity of representation is much easier to check than equality. It also saves memory, hence garbage collection time.

Terms in unification problems must be in long head-normal form before being compared. After applying the substitutions invented by imitation or projection, the flexible term may be no more in long head-normal form. However, its new long head-normal form is easy to deduce from the term and the substitution without using the β-reducer. So, imitation and projections invent a substitution value, substitute it for the head of the flexible term, compute its new long head-normal form, and substitute it for the flexible term.

For instance, unification problem $<t_1\ ,\ t_2>$, where t_1=λx(t3), t_3=(U (x S_1)), and t_2=λx(x S_2), yields three substitutions after one run of MATCH:

1. [U←λy(y)] (projection substitution),
2. [t_3←(x S_1)] (for direct long head-normalization of t_1 before passing it to SIMPL),
3. $t_1 \leftarrow t_2$ (substituting equal for equal).

Remember that unknowns, abstractions, and potential redexes are all represented by muterms. So, they are reversibly mutable. The conclusion is that much more substitutions than the *solution substitutions* are done. The supplementary substitutions contribute to saving unification and reduction time, and to saving memory.

3.2 Proof-Search

3.2.1 Prolog control

The representation of the search-stack controlling the search process uses MALI's term-stack. It is considered as a *failure continuation*. Specialized commands are defined for manipulating the failure continuation. The representation of the proof-stack controlling the development of the proof tree also uses MALI. It is mapped on compound terms. It is considered as a *success continuation*, and other commands are defined for manipulating it.

Since the term-stack and compound-terms are regular MALI's terms, we have a uniform representation of λProlog terms and control. This makes continuation capture (of both kinds) trivial [10]. It appears that implementing the Prolog cut merely requires to capture the failure continuation when entering a clause (a reification) and reinstalling it (a reflection) when executing the cut predicate. All this comes for free by using MALI.

Given program

```
in (f a) .      in (f b) .
trans (f X) (g X) .
out X :- ...
:- in I , trans I O , /*1*/ ! , out O .
```

the resolution state at label */*1*/* can be represented in MALIv06 like

```
success_continuation =
        cut_goal@(tu S_GOAL 3 (at S_SYMB cut) eos
        out_goal@(tu S_GOAL 3 (at S_SYMB out)
                O@ga@(tu S_APPL 2 (at S_SYMB g) (at S_SYMB a))
        eop@(tu S_GOAL 2 (at S_SYMB end_of_proof) (c0 S_NIL))))

failure_continuation =
        in2@(le S_CHPT
              (c2 S_ROOT (at S_INT 2)
                  fsc@(tu S_GOAL 3 (at S_SYMB in) I@(mc2 S_UNK type (at S_SIG 0))
                  (tu S_GOAL 4 (at S_SYMB trans) I O@(mc2 S_UNK type' (at S_SIG 0))
                  cut_goal)))
        eos@(le S_CHPT
              (c2 S_ROOT (at S_INT 2)
                  (tu S_GOAL 2 (at S_SYMB end_of_search) (c0 S_NIL)))
        _ ))
```

The two predicates *end_of_proof* and *end_of_search* are used for terminating the success and failure continuations. Label *O* occurs in *success_continuation* and *failure_continuation* accompanied with different terms. Terms in *success_continuation* differ by a substitution from terms with same labels in *failure_continuation*, but they share the same representation anyway. We leave unspecified the types *type* and *type'* of unknowns *I* and *O*. Note that the argument of goal *'!'* is a substack of the search-stack. Binary constructs of sort *S_ROOT* represent the roots of the choice-points. They contain a clause number and a success continuation.

After goal *'!'* is executed, the state is

```
success_continuation = out_goal
failure_continuation = eos
```

In real-life, the first goal of a success continuation is dispatched into several registers. This saves "consing" and "deconsing" the continuation.

3.2.2 Universally quantified goals

They are implemented as we have said about universal variables. The current nesting level is in fact a *signature continuation*. It has the same search-dynamism as the success continuation. This means that it is saved (i.e. pushed on MALI's term-stack) and restored (i.e. popped from MALI's term-stack) with the success continuation.

3.2.3 Implication goals

Implication is the other new construct of λProlog that enforces checking scoping conditions. The premise of an implication goal must be added to the program for the length of the proof of its conclusion.

Every premise is compiled as a clause whose logical variables are the proper logical variables of the premise, plus the logical variables of the nesting clause that occur in the premise. More formally, a dynamic clause $\forall\tilde{x}(\Phi{:-}\Psi)$, defining a predicate p, with free variables $\tilde{y}_m$ [11], is translated into the static clause $\forall\tilde{y}_m\,\tilde{x}((p'\ \tilde{y}_m\ \Phi){:-}\Psi)$, where p' is a new constant corresponding to this occurrence in the source program. The new clause is then compiled as an ordinary static clause.

Premises are *activated* when their implication goals are executed. The scope of active premises is controlled by a *program continuation* [7] that is implemented as MALI's terms, and has the same search-dynamism as the success continuation and the signature continuation. The program continuation is made of closures that enrich every active premise with their activation context. A closure links constant p with a constant p' in a context made of the current values $\tilde{c}_m$ of variables $\tilde{y}_m$. It has the form $<p,\ \lambda g(p'\ \tilde{c}_m\ g)>$.

When a goal $(p\ \tilde{s})$ is executed, the program continuation is searched for all the closures whose left-hand part is p. Their right-hand parts are successively applied to $(p\ \tilde{s})$, and the results, $(p'\ \tilde{c}_m\ (p\ \tilde{s}))$, are called. Since the abstraction in closures is very specialised (there is always one occurrence of the λ-variable g, and it is always in the same position) it need not be actually implemented as an ordinary abstraction.

For instance, in implication goal

[11] Free variables may be either universal (e.g. *pi x\(p x=>q)*) or existential (e.g. *(p X=>q)*).

```
pi Z\( p X Z (f Y) :- q W ) => G                    % X, Y and W are free.
```

the dynamic clause is compiled as

```
p_prime X Y W (p X Z (f Y)) :- q W.
```

Predicate *p_prime* must be compiled in the usual way. Among other things, its forgotten types must be computed and implicitly represented. The atom *(p X Z (f Y))* is passed as a whole in the head of *p_prime* for making the compilation easier. Another solution is to pass its parameters separately.

So, instead of storing an entire clause in the program continuation, a pair $<p, \lambda g(p\_prime\ X\ Y\ W\ g)>$ is stored every time the implication goal is called. The pair associates the predicate constant p to the new constant *p_prime* in the environment of variables X, Y, and W.

This scheme is similar to what Jayaraman and Nadathur propose [21]. The main difference is that there is only one thing to say about the interferences with backtracking: it is automatically done by MALI.

Predicates that can be extended by implication are declared *dynamic* so that not every predicate pays for implication. When a goal of a dynamic predicate is executed, one first searches the program continuation for matching premises.

3.3 A Memory Policy

The choice of a memory policy was left undefined at the level of MALI. It is still too soon to wire it at the level of the SIM because the same machine will be used in λProlog applications with totally different memory requirements (any combination of consumption rate and instantaneous working space). Since generated applications are portable, the same machine will also be used in different configurations of host systems (any combination of CPU speed and sizes of main memory and secondary memory).

We designed a memory policy which is both parameterisable and adaptative. The supplies given to MALI, the part it actually uses, and other parameters are continuously monitored, and evolution parameters are changed automatically. However, this may not be flexible enough and every executable file resulting from the compilation of a Prolog/Mali program accepts conventional arguments for configuring the memory policy to the users's will.

4 A Compilation Scheme

λProlog programs are translated into C programs which serve as a glue for putting together sequences of SIM commands. The use of C is purely incidental, but its availability and portability are good points. The C program is compiled with the regular C compiler/linker, producing an executable file for the host system. The generated C program is responsible for realising the standard interface (call/return conventions, input/output ports) with the host system.

The commands of the specialized intermediate machine are assembled so that when the generated program is executed, it has the intended proof-search behaviour. Many arrangements are possible for producing the intended behaviour. Compiling becomes really valuable when special source patterns exist for determining efficient arrangements. Efficient arrangements are in fact specializations of a general execution scheme. We list some of the patterns our compiler recognizes and the associated specializations and savings.

4.1 Special Static Patterns

4.1.1 Forgotten Types

We have seen that types must be represented at run-time. A naive solution would be to represent the types of every term and subterm. The important pattern that improves the representation of types is the occurrence of forgotten types in type declarations. They indicate the only places in which types need to be represented for checking the well-typing of unification problems.

Furthermore, the type checking/inference done at compile-time indicates which types are identical and can share representation.

4.1.2 Combinators

β-Reduction requires duplicating left members of redexes. However combinators need not be duplicated and their representation can be shared.

Since substitution values are always combinators, all instances of combinators of the source program are combinators. So, it is worth recognising them at compile-time. Our experiments show that it is a very important pattern, and that using it properly changes the complexity of programs [11].

4.1.3 First-Order Applications and Constants

The general unification procedure of λProlog is Huet's procedure augmented with dynamic type checking. However, first-order terms deserve a more direct unification procedure. So, these patterns are compiled rather classically. The representation of first-order applications is chosen to be easily recognized so that, at run-time, unification and β-reduction are improved.

4.1.4 $\beta\eta$-Normalization

Source clauses are $\beta\eta$-normalized before generation. This provides a macro-like feature which may improve the programming style. Furthermore, first-order applications are put in η-long form. This makes dynamic long head-normalization less necessary.

η-Expansion must be done carefully so that it does not create artificially large β-redexes. So, abstractions that are created by η-expansions are tagged, and β-redexes built with them are reduced using equality $(\lambda_\eta x(E\ x)\ F) \equiv_\beta (E\ F)$. New sorts are required for representing the terms of the cross-product (combinator/non-combinator)×(eta-expanded/non-eta-expanded). Again, it is a very important pattern that changes the complexity of programs [9].

4.1.5 A Weak Substitute for Clause Indexing

Clause indexing is the exploitation of the clause heads contents for computing more direct clause selection procedures. For the moment, a very simple clause indexing is implemented in Prolog/Mali.

Usually, when control enters a clause that is not the last clause of a predicate, a choice-point is created (or an already existing choice-point is updated). It can be a waste of time and memory if a succession of choice-point creations and choice-point consumptions is used to select a clause in a predicate. Clause indexing helps selecting more directly the proper clause.

The lack of clause indexing is somewhat compensated by delayed creation of choice-points. Delayed creation of choice-points amounts to indicating that a choice-point is to be created instead of creating it. The creation must be resumed as soon as a logical variable is substituted, or when unification succeeds (if no logical variable is substituted). If a failure occurs while the choice-point creation is still delayed, failure is merely implemented as a jump.

More interestingly, substitutions of a head-normal-form to a non-normal form do not count as substitutions of logical variables. So, they do not trigger the choice-point creation. The neat effect is that a goal argument will be reduced only once for all the attempts at unifying a clause head, whereas if the choice-point were created as soon as ordered then the goal argument would be reduced for every unification attempt, and unreduced at every backtrack. For instance, in

```
test 0 :- do_something .
test 1 :- do_something_else .
query :- big_redex N , M = 1, test (N x\x M) .
```

the "big redex" N is reduced only once instead of twice.

The brute force solution consisting in reducing a goal before unifying kills lazyness, and does not eliminate the need for normalising during unification because substitutions might build redexes. So, delayed creation of choice-point gives a partial solution to a critical problem that appears every time normalization of terms or awakening of constraints are possible.

4.2 The Translation

λProlog programs are translated into C on a predicate-to-function basis. Every predicate is implemented as a function of the continuations (success, signature, program, and failure) that returns new values for the continuations.

The functions never call each other; recursion is taken into account by the success continuation. Functions are called by, and return to, a *motor*, which can be considered the last remnant of an interpreter. Some static patterns, such as left-recursion, allow to avoid going through the motor.

As we have seen in section 3.1.1, type schemes are translated into type reconstruction functions. Furthermore, every constant (predicate and function constants, and type constructors) is translated into a C structure containing their external representation, their arity, their predicate function or type reconstruction function, if needed, and any useful information.

5 Conclusions

5.1 Prolog/Mali

The Prolog/Mali compiler is written in λProlog and the run-time libraries are written in λProlog and in C. The Prolog/Mali system has been completely bootstrapped. It implements all the core of λProlog plus various extensions. One of the most notable extensions is the continuation capture capability. It is used for implementing the cut and a catch/throw escape system [10].

Prolog/Mali is freely available, and used in several research teams in domains such as automated theorem proving, automated learning, and meta-programming. Some of the benchmarks used for comparing Prolog/Mali with other implementations come from these teams.

5.2 Comparisons with Other Works

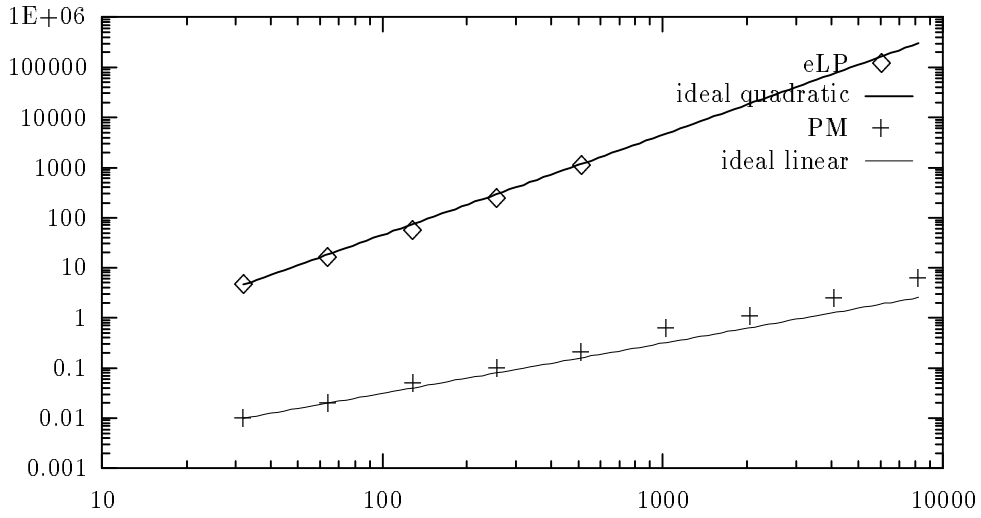

Figure 1: Comparison of time complexities for reversing a function-list (log-log scale; X-axis: list-length; Y-axis: run-times in seconds)

It has not been possible to compare our system with the other most recent attempts for implementing λProlog (Nadathur, Jayaraman, Felty), because of the lack of availability of a complete system. However, papers and technical reports by Nadathur and Jayaraman [35, 22, 21] show that their approach and ours are somewhat different and difficult to compare on the paper. In few words, they choose to base their design on a WAM augmented for handling λProlog's specifics. They represent λ-terms and reduction in an environment-based fashion. However, the differences may be blurred by optimizations that apply techniques from one paradigm for improving the other.

A technical report by Kwon, Nadathur and Wilson [24] proposes a handling of types at run-time which is similar to ours, except that forgotten types are not the only types represented in constants. These authors and

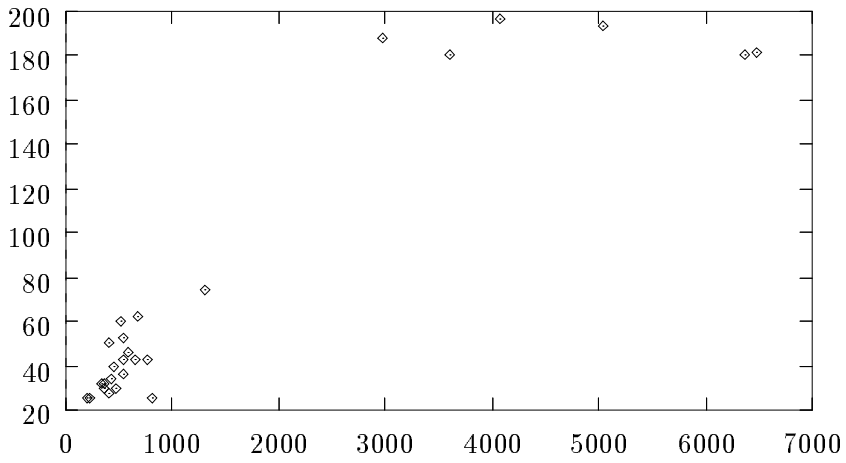

Figure 2: Comparison of run-times for executing a tactical theorem prover (X-axis: Prolog/Mali run-times in milliseconds, Y-axis: Prolog/Mali / eLP speed-ratio)

Jayaraman's basic technical choice is to extend a structure-sharing implementation of the WAM: it also applies to the representation of types.

The only λProlog system with which we have made extensive comparisons is eLP. It is already an "old" system. eLP is an interpreted system written in Lisp. The fact that it is interpreted could have explained a constant speed factor between eLP and Prolog/Mali. However, what is observed is a difference in complexity that interpretation costs cannot explain alone. We compared Prolog/Mali and eLP in a black-box mode, knowing nothing of the implementation of eLP. The comparison has been done using special purpose programs for exhibiting qualitative differences, and also using regular programs from λProlog users.

The memory management improvement over eLP is dramatic for any kind of program. It is also better than many implementations of standard Prolog. The Lisp system that supports eLP has its own memory management, which might be efficient as far as Lisp evaluation is concerned. But it does not know about logic programming usefulness logic, and does nothing when early reset and muterm shunting are in order. It is a definitely bad idea to leave a non logic programming system in charge of logic programming memory management. This does not forbid implementing logic programming in a foreign language; the only thing is that logic programming memory management has to be redone in that language.

Special purpose programs show an arbitrary speed-up of Prolog/Mali over eLP's. The complexity of both unification and reduction is higher in eLP. The systematic sharing and folding of representations, and the detection of combinators and η-redexes play a critical part in the better complexity of Prolog/Mali. Delaying the creation of choice-points also improves the complexity of search. Figure 1 shows the behaviours of eLP and Prolog/Mali when executing the program that naively reverses a function list. Scales are logarithmic on both axes. Continuous lines correspond to the ideal linear or quadratic case. The slopes of the lines, 1 and 2, indicate a linear complexity for the first and a quadratic complexity for the second.

Regular programs (mainly a demonstrator with tacticals, and a demonstrator with a learning component) show a speed-up between 25 and 250. Interestingly enough, for a given program, the speed-up grows with the time required for executing a query. This shows that eLP does not scale up very well. Figure 2 shows the speed-up of Prolog/Mali over eLP for a set of small theorem proving problems. Every point correspond to a particular problem.

Finally, we compared Prolog/Mali with modern implementations of Prolog. For real size programs (e.g. an early version of our compiler), Prolog/Mali is less than 10 times slower than Prolog (≈ 5 on the average). For instance, the *chat parser* benchmark runs 6 times slower with Prolog/Mali than with SICStus Prolog. This comparison is a little bit unfair for Prolog/Mali, and for λProlog in general, because it executes the first-order Horn clauses fragment of λProlog with a higher-order hereditary Harrop formulas technology. When what the user requires is exclusive to λProlog, the standard Prolog programmer has to implement it at the Prolog level; it is certainly less efficient, and less safe too, than what a λProlog system offers.

5.3 Further Work

Although our implementation of λProlog enjoys nice complexity properties, and its performances are encouraging, it is rather slow when it is compared with the current state of the art for standard Prolog. In its present state the control of search is compiled but unification of higher-order terms is not and there is little clause indexing. Our current implementation task is to devise a compilation scheme for unification and indexing so as to bring the performance level of the standard part closer to the current state of the art.

To improve performances, more static analysis ought to be performed. For instance, it is important to detect when the full mechanism of Huet's unification is not needed. The L_λ [27] fragment of λProlog has a unitary and decidable unification theory. Belonging to L_λ is easy to test at run time but it could be more efficient to detect that some predicate or some argument will always be in L_λ. Note that the L_λ property generalizes every pattern that the TRIV procedure currently recognizes.

5.4 Remarks on Focusing on Memory Management

Our main implementation concern has been memory management. We always tried to have memory management problems solved before time efficiency problems. This is reflected in the software architecture of Prolog/Mali, in which the kernel (MALI) knows almost everything about memory management but nothing on the procedures that will be used, the specialized abstract machine knows less about memory management, and a little bit more about the procedures, and the generated code knowns about the procedures (it is part of them) but is really naïve as far as memory management is concerned. However, a reasonable time efficiency has been achieved, and still more can be gained with further efforts.

Two keywords of this implementation are *sharing* and *folding* of representations. Sharing amounts to recognising that some representation already exists and reusing it. Folding amounts to recognising that two different representations represent the same thing and replacing one by the other.

This architecture can be used for implementing many other kinds of logic programming systems. It cannot compete for implementing standard Prolog systems because very efficient and specialized techniques have already been designed. It is perfectly fit as soon as complex data-structure and control are in order. An implementation redoing a specialized version of Mali's memory management from scratch could always be faster but will certainly be much more complex.

References

[1] M. Abadi, L. Cardelli, B.C. Pierce, and G.D. Plotkin. *Dynamic Typing in a Statically Typed Language.* Technical Report, DEC Systems Research Center, 1989.

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

[3] J.-M. Andreoli and R. Pareschi. Linear objects: logical processes with built-in inheritance. In D.H.D. Warren and P. Szeredi, editors, *7th Int. Conf. Logic Programming*, MIT Press, 1990.

[4] K. Appleby, M. Carlsson, S. Haridi, and D. Sahlin. Garbage collection for Prolog based on the WAM. *CACM*, 31(6), 1988.

[5] H. Barendregt. Introduction to generalized type systems. *J. Functional Programming*, 1(2):125–154, 1991.

[6] Y. Bekkers, B. Canet, O. Ridoux, and L. Ungaro. MALI: a memory with a real-time garbage collector for implementing logic programming languages. In *3rd Symp. Logic Programming*, IEEE, 1986.

[7] P. Brisset. *Compilation de λProlog.* Thèse, Université de Rennes I, 1992.

[8] P. Brisset and O. Ridoux. The architecture of an implementation of λProlog: Prolog/Mali. In *Workshop on λProlog*, Philadelphia, PA, USA, 1992. ftp: //ftp.irisa.fr/local/lande.

[9] P. Brisset and O. Ridoux. *The Compilation of λProlog and its execution with MALI.* Technical Report 687, IRISA, 1992. ftp: //ftp.irisa.fr/local/lande.

[10] P. Brisset and O. Ridoux. Continuations in λProlog. In D.S. Warren, editor, *10th Int. Conf. Logic Programming*, pages 27–43, MIT Press, 1993.

[11] P. Brisset and O. Ridoux. Naïve reverse can be linear. In K. Furukawa, editor, *8th Int. Conf. Logic Programming*, pages 857–870, MIT Press, 1991.

[12] M. Bruynooghe. Garbage collection in Prolog implementations. In J.A. Campbell, editor, *Implementations of Prolog*, pages 259–267, Ellis Horwood, 1984.

[13] C.M. Elliott. Higher-order unification with dependent function types. In N. Derschowitz, editor, *3rd Int. Conf. Rewriting Techniques and Applications, LNCS 355*, pages 121–136, Springer-Verlag, 1989.

[14] C.M. Elliott and F. Pfenning. A semi-functional implementation of a higher-order logic programming language. In P. Lee, editor, *Topics in Advanced Language Implementation*, pages 289–325, MIT Press, 1991.

[15] A. Felty. *Specifying and Implementing Theorem Provers in a Higher-Order Logic Programming Language.* PhD Dissertation, Dept. of Computer and Information Science, University of Pennsylvania, 1989.

[16] A. Felty and D.A. Miller. *Encoding a Dependent-Type λ-Calculus in a Logic Programming Language.* Rapport de recherche 1259, Inria, 1990.

[17] A. Felty and D.A. Miller. Specifying theorem provers in a higher-order logic programming language. In E. Lusk and R. Overbeek, editors, *CADE-88, LNCS 310*, pages 61–80, Springer-Verlag, 1988.

[18] M. Hanus. Horn clause programs with polymorphic types: semantics and resolution. *Theoretical Computer Science*, 89:63–106, 1991.

[19] J.S. Hodas and D.A. Miller. Logic programming in a fragment of intuitionistic linear logic. In G. Kahn, editor, *Symp. Logic in Computer Science*, pages 32–42, Amsterdam, The Netherlands, 1991.

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

[21] B. Jayaraman and G. Nadathur. Implementation techniques for scoping constructs in logic programming. In K. Furukawa, editor, *8th Int. Conf. Logic Programming*, pages 871–886, MIT Press, 1991.

[22] B. Jayaraman and G. Nadathur. Implementing λProlog: a progress report. In *2nd NACLP Workshop on Logic Programming Architectures and Implementations*, MIT Press, 1990.

[23] A.J. Kfoury, J. Tiuryn, and P. Urzyczyn. *The Undecidability of the Semi-Unification Problem.* Technical Report BUCS 89-010, Boston University, 1989.

[24] Keehang Kwon, G. Nadathur, and D.S. Wilson. *Implementing Logic Programming Languages with Polymorphic Typing.* Technical Report CS–1991–39, Dept. of Computer Science, Duke University, 1991.

[25] T.K. Lakshman and U.S. Reddy. Typed Prolog: a semantic reconstruction of the Mycroft-O'Keefe type system. In *Int. Logic Programming Symp.*, pages 202–217, 1991.

[26] S. Le Huitouze. A new data structure for implementing extensions to Prolog. In P. Deransart and J. Małuszyński, editors, *2nd Int. Work. Programming Languages Implementation and Logic Programming, LNCS 456*, pages 136–150, Springer-Verlag, 1990.

[27] D.A. Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. In P. Schroeder-Heister, editor, *Int. Workshop on Extensions of Logic Programming, LNAI 475*, Springer-Verlag, New York, 1989.

[28] D.A. Miller. A logical analysis of modules in logic programming. *J. Logic Programming*, 6(1–2):79–108, 1989.

[29] D.A. Miller. A theory of modules for logic programming. In *Symp. Logic Programming*, pages 106–115, Salt Lake City, UT, USA, 1986.

[30] D.A. Miller and G. Nadathur. Higher-order logic programming. In E. Shapiro, editor, *3rd Int. Conf. Logic Programming, LNCS 225*, pages 448–462, Springer-Verlag, 1986.

[31] D.A. Miller and G. Nadathur. A logic programming approach to manipulating formulas and programs. In S. Haridi, editor, *IEEE Symp. Logic Programming*, pages 379–388, San Francisco, CA, USA, 1987.

[32] D.A. Miller, G. Nadathur, and A. Scedrov. Hereditary Harrop formulas and uniform proof systems. In D. Gries, editor, *2nd Symp. Logic in Computer Science*, pages 98–105, Ithaca, New York, USA, 1987.

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

[34] G. Nadathur. *A Higher-Order Logic as the Basis for Logic Programming.* Ph.D. Thesis, University of Pennsylvania, 1987.

[35] G. Nadathur and B. Jayaraman. Towards a WAM model for λProlog. In E.L. Lusk and R.A. Overbeek, editors, *1st North American Conf. Logic Programming*, pages 1180–1198, MIT Press, 1989.

[36] G. Nadathur and D.S. Wilson. A representation of lambda terms suitable for operations on their intensions. In *ACM Conf. Lisp and Functional Programming*, pages 341–348, ACM Press, 1990.

[37] T. Nipkow. *Higher-Order Unification, Polymorphism, and Subsorts.* Technical Report 210, University of Cambridge, Computer Laboratory, 1990.

[38] F. Pfenning. Logic programming in the LF logical framework. In G. Huet and G. Plotkin, editors, *Logical Frameworks*, pages 149–181, Cambridge University Press, 1991.

[39] F. Pfenning. Unification and anti-unification in the calculus of constructions. In *Symp. Logic in Computer Science*, pages 74–85, 1991.

[40] O. Ridoux. *MALIv06: Tutorial and Reference Manual.* Publication Interne 611, IRISA, 1991. ftp: //ftp.irisa.fr/local/lande.

[41] D. Sahlin and M. Carlsson. *Variable Shunting for the WAM.* Research Report SICS/R–91/9107, SICS, 1991.

[42] D.H.D. Warren. *An Abstract Prolog Instruction Set.* Technical Note 309, SRI International, Stanford, Ca, 1983.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 46 avenue Félix Viallet, 38031 GRENOBLE Cedex 1
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399