Programming with MALI. The interpretation of Prolog programs

Louis Chevallier, Serge Le Huitouze, Olivier Ridoux

INRIA

UNITÉ DE RECHERCHE
INRIA-RENNES

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél. (1) 39 63 55 11

Rapports de Recherche

N° 1048

Programme 2

PROGRAMMING WITH MALI THE INTERPRETATION OF PROLOG PROGRAMS

Louis CHEVALLIER
Serge LE HUITOUZE
Olivier RIDOUX

Juin 1989

RR-1048

IRISA

INSTITUT DE RECHERCHE EN INFORMATIQUE
ET SYSTÈMES ALÉATOIRES

Campus Universitaire de Beaulieu
35042 - RENNES CÉDEX
FRANCE
Téléphone : 99 36 20 00
Télex : UNIRISA 950 473 F
Télécopie : 99 38 38 32

Publication Interne n° 470 - Mai 1989 - 16 Pages

Programming with MALI — The Interpretation of Prolog Programs [1]

Louis Chevallier — Serge Le Huitouze — Olivier Ridoux

Abstract

MALI is a logic programming machine, equipped with an automatic memory management. We present the functionalities of MALI to describe the influence of a trustworthy garbage collector on the implementation of logic programming languages and their usage. We show how to implement logic programs on this machine, and argue that it is simpler and in certain cases more efficient than in a more classical way. We also argue that the logic programming style can be improved since there is no need for inserting garbage collecting tricks in logic programs. Finally, we show how it helps to introduce new features that are otherwise difficult to implement because of their memory requirements.

Programmer avec MALI — l'interprétation des programmes Prolog

Résumé

MALI est une machine de programmation logique, munie d'une gestion de mémoire automatique. Nous présentons les fonctionnalités de MALI et l'influence d'un récupérateur de mémoire efficace sur l'implémentation des langages de programmation logique et leur usage. Nous montrons comment implémenter la programmation logique sur cette machine, et plaidons que cela est plus simple et parfois plus efficace qu'avec une méthode classique. Nous défendons aussi que le style de programmation en logique peut être amélioré en supprimant le besoin d'introduire des astuces de récupération de mémoire dans les programmes en logique. Finalement, nous montrons comment cela permet d'introduire de nouveaux dispositifs qu'il est difficile d'implémenter autrement à cause de leur consommation de mémoire.

[1] A previous version of this paper has been presented at the "joint CWI — GMD — INRIA workshop" (April 1989, St. Augustin, Schloß Birlinghoven, Federal Republic of Germany).

CENTRE NATIONAL DE LA RECHERCHE SCIENTIFIQUE
(L. A. 227)
UNIVERSITÉ DE RENNES 1 I. N. S. A. DE RENNES

INSTITUT NATIONAL DE RECHERCHE
EN INFORMATIQUE ET EN AUTOMATIQUE
(LABORATOIRE DE RENNES)

1 Introduction

Time and space eagerness are the major handicaps of non-side-effect programming. Time is less crucial since the handicap is often a constant factor that can be decreased by further improvement in computing scheme or hardware. The space handicap cannot be limited to a constant factor since it is roughly a linear function of time. The reason is that the non-side-effect symmetric of assignment is the action of creating the representation of a new value. Garbage collection is the routine that, given some knowledge of the semantics of the language, detects data structures which represent values that are no more in use, and frees the space they occupy. For a side-effect language, the assignment of a new value to a memory forgets automatically the ancient value; it is the very simple memory management implied by assignment. Garbage collection is the well behaved symmetric of forgetting of the ancient value for non-side-effect languages.

As a non-side-effect programming paradigm, logic programming (and Prolog) requires garbage collection. The classical Prolog solution is to avoid garbage collection, adopting an interpreting scheme where some memory management is performed. This scheme is based on the procedural semantics of Prolog and the observation that, when deterministic, Prolog programs requirements are the same as procedural programs requirements (a stack and a heap). The advantage of this method is that it is applicable in the situation where efficient compilation is possible (Prolog strategy resolution). The drawbacks are: it is not complete since the heap must be garbage collected; it requires a priori determinism detection to be efficient, though most of the determinism comes a posteriori from the 'cut'; it is tightly bound to a particular interpretation scheme, namely the Warren Abstract Machine (WAM) [War83].

An almost as classical solution is to settle Prolog on the top of a system supplied with a garbage collector, commonly Lisp. This solution is not so good in that memory requirements of Prolog are not the same as Lisp's. We call usefulness logic the property of the data structures that are deemed useful. Lisp's usefulness logic is connexity with some roots. Due to non-determinism, Prolog's usefulness logic is much more demanding. A data structure is useful if it represents part of an active goal statement [BCRU84a, PBW85] (active goal statements are the current goal statement and backtrack points). So a Prolog program still performs badly on a Lisp system.

Our solution is to design garbage collection for logic programming independently from any interpreting scheme. For that, we need to restrict the behaviour of future interpreter with the definition of the set of actions to be used. We shall present these actions in section 2, and the way we use them in section 3. Finally, we argue in section 4 that the benefits for the Prolog programmer are the same as those for the interpreter designer.

2 MALI: a logic programming machine

MALI [BCRU84b, BCRU86] (Machine Adaptée aux Langages Indéterministes — machine fitted for non-deterministic languages) is an abstract machine whose semantics are well-suited for an implementation with a complete parallel real-time garbage collector. It implements a usefulness logic that has been shown specific for depth-first search languages [Rid86]. It behaves like a memory with much more commands than the only 'read' and 'write'. MALI has already a hardware implementation on a bi-processor architecture, and a software implementation written in C.

Since depth-first search is in the focus of MALI, everything that has to do with it must be defined at MALI level (i.e. backtrack point setting and deleting, variable creation and binding, and backtracking). So everything such as backtrack stack and trail is encompassed in MALI and is not of the designer's responsibility, thus simplifying his work. Furthermore, there is a term construction and traversal capability which provides the kernel of value handling for the user.

In the following, identifiers of MALI's specific concepts are *emphasized.*

2.1 Basic commands

The commands of MALI have arguments of a specific type called *Names*, and occasionally integers. A partial function, the *Designation*, associates a *Term* or a stack of *Terms* to a *Name*. So, *Name* and *Designation* are the MALI correspondents to random memory address and contents.

A *Name* contains a tag which indicates the *Nature* and *Sort* of a *Term*. The *Nature* is either *Atom*, *Construct*, *Tuple*, *Variable*, *Attributed Variable* or *Level*. The *Sort* gives elementary typing to *Atoms*, *Constructs*, *Tuples*, *Variables* or *Attributed Variables*.

We give the semantics of MALI through its commands. Their signature is given as

`Function` `(I1,...,In) -> (O1,...,Om)`

where the `Ii`'s (resp. `Oj`'s) are the input (resp. output) parameters types. First letters of *Nature* identifiers (A, C, T, V, AV, L) are prepended to *Name* or *Sort* to constrain the type of inputs or outputs.

A command is designed to analyse *Names*.

`NatureSort` `(Name) -> (Nature,Sort)`

It gives the *Nature* and *Sort* of a named *Term*.

2.1.1 The state

The state of MALI is composed of the *Designation* and the *Search Stack* which is a stack, the elements of which are *Terms*. The *Designation* and the *Search Stack* are two separate and independent components of the state. In the following, we only expose facts that are relevant to the state evolution. As a consequence, when a component of the state is left unchanged by the execution of a command, nothing is said about this component.

2.1.2 Data structure

The user knows the structure of an *Atom Name*, and he builds bits patterns that look like *Atom Names* for representing his own atomic values. All other *Names* are synthesised by MALI itself.

The following commands are intended to be used for compound value representation.

`NewCons` `(C_Sort,Name,Name) -> (C_Name)`
`RightSubCons, LeftSubCons` `(C_Name) -> (Name)`

The result of `NewCons` is a *Name* of the consed *Term* whose sub-*Terms* are the *Terms* that are designated by the two last parameters. Access to sub-*Terms* is done with `RightSubCons` and `LeftSubCons`. All this is very similar to 'cons', 'car' and 'cdr' in Lisp. Similar commands create *Tuples* and access their elements.

2.1.3 Variables

Executing the preceding commands only extends the domain of the *Designation*. Other commands allow to modify the *Designation* at some points of its domain.

These commands define the use of *Variables*. *Variables* are analogous to the logical variables of Prolog.

`NewVar` `(V_Sort) -> (V_Name)`
`BindVar` `(V_Name,Name) -> ()`

Function `NewVar` returns a *Name* of a new *Variable*. Function `BindVar` substitutes a *Term* to a *Variable*. Every *Name* that designates the *Variable* before the substitution, designates the *Term* after. An effect of a substitution is that several *Names* designate the same *Term*.

`NewAttributedVar`	`(AV_Sort,Name) -> (AV_Name)`
`BindAttributedVar`	`(AV_Name,Name) -> ()`
`GetAttribute`	`(AV_Name) -> (Name)`

Function `NewAttributedVar` returns a *Name* of a new *Attributed Variable*. An *Attributed Variable* works like a *Variable*, except that it has an associated value: a *Term* called *Attribute*. Function *GetAttribute* returns a *Name* of the *Attribute* of an *Attributed Variable*. We give in 3.3.2 and 3.4.2 two examples of the use of *Attributed Variable*.

2.1.4 Search control

The following commands are a generalisation of the depth-first search strategy of Prolog. They handle the *Search Stack*.

`Save`	`(Name) -> ()`
`Retrieve`	`() -> (Name)`

Function `Save` pushes the named *Term* on the *Search Stack*. It is used to set a backtrack-point. Conversely, `Retrieve` pops a *Term* from the *Search Stack* and returns a *Name* that designates it. It corresponds to a backtrack. The *Designation* after a retrieval is only specified for the resulting *Name*. The user has to traverse the retrieved *Term* to get further information.

`CurLevel`	`() -> (L_Name)`
`Cut`	`(L_Name) -> ()`

Function `CurLevel` gives a *Name* that designates the current value of the *SearchStack*. Conversely `Cut` sets the *Search Stack* to the designated value. This command corresponds to the Prolog 'cut'. Every designated stack value must be a sub-stack of the *Search Stack*. So, at the execution of `Cut`, every name that designates a stack that contains the given one, is set to designate this one.

2.1.5 Garbage collection control

MALI is defined in such a way that an efficient garbage collector is easily implemented. The theory of MALI memory management shows the garbage collector as a routine that the user calls with a *Name* of the useful *Term* as a parameter.

`Reduce`	`(Name) -> (Name)`

This command makes the designated *Term* to be the only useful *Term* of the *Designation*. The *Designation* is reduced (i.e. its domain is reduced) to the resulting *Name*. This *Name* designates the only useful *Term*.

The algorithmic effect of `Reduce` depends on the implementation of MALI, but is almost never an actual procedure call. We present two cases.

1. The collector is a sequential routine, triggered near exhaustion. So, `Reduce` has no effect if a given occupancy threshold is not reached. Otherwise, the garbage collection routine is first executed, and then `Reduce` gives control back.

2. The collector is a parallel process. So, `Reduce` has no effect if the collecting process is already executing. Otherwise, an interaction triggers its execution, and `Reduce` gives control back without waiting the end of the collection.

Hence the collector does not automatically iterate its routine; it must wait a signal (`Reduce`) from the MALI user. Thanks to this feature, the user can keep as many *Names* as he wants in his private memory, but must signal from time to time to the collector the *Name* of the *Term* he actually uses, for the collector being able to operate.

2.1.6 Garbage collection

We shortly present the features of MALI memory management. Hence, we must describe how the *Designation* is represented in a memory.

Construct, *Tuples*, *Variables*, *Attributed Variables* and *Levels* are represented by records, the references thereof are in their *Names*. Depending on the *Nature* of *Terms*, the record fields contain the *Names* of sub-*Terms*, binding value, *Attributes*, accesses to the saved *Terms* or trailing information.

The record that represents a *Variable* of the *Designation* has a binding value field that contains the mark 'unbound'. Substituting a *Term* to a *Variable* (*Attributed* or not) amounts to assign a *Name* of the *Term* to the binding value field.

The *Search Stack* is represented by the chaining of the *Levels* records. When saving a *Term* an access to its representation is stored. The fact that no new representation is needed is called 'or-sharing'. Since binding a *Variable* amounts to a side-effect in the *Term* representation, and the saved *Terms* as well, it must be remembered to be cancelled when executing **Retrieve**. It is the job of the trail to remember bindings.

To sum up, the *Designation* and the *Search Stack* consume records for their representation. The collector task is to detect records that do not contribute to their representation.

The ways for a record to become useless are many but can be classified in two categories.

1. The execution of **Reduce**: the representation of *Terms* that are neither sub-*Terms* of the **Reduce** argument nor sub-*Terms* of a saved *Term* is useless.

2. The execution of **Retrieve** or **Cut**: the representation of stack values (trail included) deleted by these commands is useless if it is not used by other stack values or the *Designation*.

All the novelty of Prolog-like memory management comes from the fact that the evolution of the *Designation* is constrained by the depth-first strategy. The garbage collector has to interpret the representation of *Variables* according to the *Search Stack* and the trail. This novelty is entirely accounted for in MALI.

For instance, the collector detects that the binding of a *Variable* is useless. It compares the representations of the binding and the saved *Terms*. Conventionally, a saved *Term* is older than an other if it closer to the bottom of the *Search Stack*, and a binding is newer than every saved *Term* in which it is not in effect. Given a bound *Variable* record, if every *Term* whose representation uses this record is older than its binding, then the binding is useless. The collector assigns the mark 'unbound' to the binding value field to delete the binding. So, bindings can be undone before the execution of **Retrieve**; this is the 'binding-shunting' capability of MALI. It is needed for programming a 'perpetual process' as in [War82]. This feature is called 'virtual backtracking' in [PBW85]

The collector also detects that a *Variable* record is useless though the binding value that it contains is useful. Given a bound *Variable* record, if every *Term* whose representation uses this record is newer than its binding, then the *Variable* is definitively bound and its binding value will be physically substituted to every occurrence of the *Variable*. So, the *Variable* records that are merely scaffoldings are removed when the construction is finished; this is the 'variable-shunting' capability of MALI. These records commonly cover about half or two thirds of the memory.

Finally, the specification of MALI (either software or hardware) accepts a collection routine that has a linear time-complexity with the number of useful records. This property which is trivial in the case of Lisp, is not so manifest in the case of depth-first search, and is not satisfied by non-depth-first search [Rid86].

2.2 Advanced commands

Advanced commands are supplied to fill the gap between the semantics of MALI as described by previous commands, and the intended use, that is logic programs execution.

The following properties cause the gap.

- Bottom-up construction of *Terms*. One must construct a *Term* from the leaves to the root. But top-down or breadth-first constructions are desirable and more efficient since they allow the Term in construction to serve as its own construction stack.
- Record-like access to *Terms*. One must use a stack to fully traverse a term. It can be a bottle-neck for demanding routines such as unification. Here too, top-down traversal is desirable.
- No side-effect on *Terms*. It is a primary property of MALI since *Terms* are values. These values are to be used to represent user level-terms. Efficient unification algorithms use to do side-effects on the representation of user level-terms. So, they have to do side-effects on *Terms*.

All these problems deserve specialised solutions which are not in the scope of this paper. However, it is important to note that the basic principles of MALI need not be altered. So the previous description is semantically complete.

3 Executing logic programs with MALI

In this section we show how to use MALI commands to execute logic programs. A part of the programming style inherited from memory management manifests itself at this level, the other part at the Prolog programming level. Prolog is taken as the paradigm of logic programming, but some extensions are explored.

The general architecture is the following:

- a host system stores the clauses base and the interpreter or the compiled code,
- MALI stores the resolution state —all the active goal statements— represented by *Terms*,
- the interpreter executes a loop whose step is an inference: at each step the **Reduce** command is executed with the current goal statement. Doing so, it explicitly prompts the current goal statement to the collector, and implicitly the saved goal statements.

3.1 Goal statements

The standard 'Copy vs Structure-sharing' debate is somewhat biased when using MALI, because the basic representation material is the *Term* instead of linear memory. We first expose our vision of the debate and the MALI's bias and then how we represent goal statements in MALI.

3.1.1 Copy vs Structure-sharing

The resolution principle [Rob65] specifies that the unification of two complementary goals must be done modulo the renaming of their variables, in order to avoid clashes of variable names. In the case of Prolog, this amounts to rename the variables of the clause chosen to solve a selected goal. We call this operation 'exemplarisation of a clause'. In the following, we do not matter how the clause is chosen or the goal selected, nor on the management of the search space. We only study the renaming representation, and the basic operations of the resolution step. The chosen clause is supposed to be represented in the clause base, and the selected goal in MALI. The operations to be performed are the variables renaming, the unification of the goal and the clause head (with application of the substitution in the clause body and in the remaining goals), and the addition of the clause body to the remaining goals.

The simplest way to perform these operations (see Figure 1) is to create a representation of the clause in MALI in which every Prolog variable is represented by a new *Variable*. Then the head of the copy and the goal are unified, and if unification succeeds the body of the copy is appended to the remaining goals to form the new goal statement. The application of the substitution is done

```
{ Term CCC ;
  CCC = Copy of the Chosen Clause ;
  Unify the Head of CCC with the Selected Goal ;
  Append the Body of CCC to the remaining Goals ; }
```

Figure 1: Naive Copy

with **BindVar**. Though simple, this is naive and inefficient because many copies may be useless due to unification failure. However, meta-interpreters written in Prolog with the predicate 'clause' work this way. New and classical implementation of renaming can be described by the amount of admitted copy, or by the extent to which the copy is delayed. The copy procedure needs a local context to associate the same *Variable* to all occurrences of the same Prolog variable. We shall see that delaying the copy stretches the scope of this context.

```
{ Term CHCC, CBCC, CTXT[] ;
  CHCC = Copy of the Head of the Chosen Clause( CTXT ) ;
  Unify CHCC with the Selected Goal ;
  CBCC = Copy of the Body of the Chosen Clause( CTXT ) ;
  Append CBCC to the remaining Goals ; }
```

Figure 2: Less Naive Copy

The first improvement (see Figure 2) is to copy only the head before unification and copy the body on success. The context is no more local because it is the same for the two copying phases. However, it remains local to the resolution step. We do not know any real-life system that uses this strategy, but we believe that it is a necessary step for our exposition.

```
{ Term CBCC, CTXT[] ;
  (  Unify the Head of the Chosen Clause
           with the the Selected Goal( CTXT )
  // Copy SubTerms of the Head( CTXT )
  ) ;
  CBCC = Copy of the Body of the Chosen Clause( CTXT ) ;
  Append CBCC to the remaining Goals ; }
```

Figure 3: Smart Copy

The copy of the head is only needed for the parts of it that happen to be binding values of goal variables, because what is substituted to a *Variable* must be a *Term*. So, a further refinement (see Figure 3) is to copy these parts of the head only when substitutions are applied. The context, still local to the resolution step, is common to every invocation of the copy procedure. It is the strategy of all our implementations of Prolog with MALI. It requires two instances of the unification procedure: one for two terms represented in MALI (we call it dynamic unification), the other when a term is not represented in MALI (static unification). The latter may call the former. Each subterm of the head can either be used by the static unification procedure or the copy procedure. These two procedures are very similar in form, and can be built by structural induction on the terms. They can even be one procedure working in two modes like the 'unify' instructions of the WAM [War83]. It traverses a

Term when in 'read' mode, and performs exemplarisation in 'write' mode. It can still be argued that some copy of clause body can be useless. This problem must be precisely evaluated but it seems that a lot of clause bodies are not bigger than a context. Moreover, copy can be postponed by a special treatment of first goals of clauses. The idea is to include the resolution of the first goals of the body in the current resolution step. One condition to perform this optimisation is that these goals do not incur recursive resolution. Built-in predicates meet this condition.

```
{ Term CTXT[] ;
  (  Unify the Head of the Chosen Clause
           with the the Selected Goal( CTXT )
  // Copy SubTerms of the Head( CTXT )
  ) ;
  Append <Chosen Clause, CTXT> to the remaining Goals ; }
```

Figure 4: Structure-sharing with copy of binding values

One can still delay the copy. Next step (see Figure 4) is to completely avoid the copy of the body, and to store instead the context and an identification of the goals of the body. Now the context becomes global. It is the strategy used in the then innovative 'Edinburgh' implementation [War77]. The contexts were stored in the 'local stack' and the binding values in the 'global stack'. This works well thanks to the left-to-right proof strategy which makes the identification of the goals of the body feasible with one pointer. A less classical strategy is painful to implement with this scheme or the next one.

```
{ Term CTXT[] ;
  Unify the Head of the Chosen Clause
        with the the Selected Goal( CTXT ) ;
  Append <Chosen Clause, CTXT> to the remaining Goals ; }
```

Figure 5: Structure-sharing

The ultimate step is to do no copy at all (see Figure 5). This is the strategy used by the first implementations of Prolog, and known as 'structure-sharing'.

The five steps of our exposition are ordered by decreasing amount of admitted copy. It is clear that the first three are also ordered by increasing efficiency (less copy and less garbage collection). The ordering of the last three is debatable as an instance of the 'Copy vs Structure-sharing' question. The novelty with MALI is that it introduces a bias towards copy since the efficient representation of contexts is in conflict with efficient garbage collection. To randomly access context entries requires linear memory, which prevents to destroy useless entries. Another bias is that the traversal of *Variables* bindings is far more efficient in MALI than it is with contexts represented in MALI. The reason of the efficiency difference is the same as between interpretation and meta-interpretation.

3.1.2 Standard Prolog strategy

Any abstract description of standard interpretation of Prolog (e.g. [Van82]) is suitable to design an interpreter using MALI. As we have exposed previously, all our Prolog implementations with MALI use copy to perform exemplarisation, but delay it until the application of substitution or the construction of the new goal statement really needs it.

This is a rather new proposition since the classical copy debate is about binding values. There is no more static structure-sharing (rather improperly shortened to structure-sharing), but only dynamic structure-sharing. Static structure-sharing deals with the use of program pieces to represent the goal statements. Dynamic structure-sharing deals with the use of the same representation by several dynamic terms.

The sharing of dynamic structures is easy and secure. It is secure in two respects: shared data structures are not discarded as long as they are usefull, and they will be discarded in a finite time as soon as they become useless. For example our treatment of TRO [War80] is the following: each procedure call (deterministic or not) naively erases the calling goal; it is up to the or-control to save a goal statement where a non-deterministic call is done. In this way, every return of procedure behaves as if TRO applied. So, the treatment of call and return is uniform. If the call is, or becomes, deterministic then the data structures representing its evaluation become garbage and are eventually collected.

3.1.3 Non-standard strategy

Non-left-to-right strategy can be implemented by replacing the list of goals by a difference-list (to obtain side-tracking), or by a labelled tree where labels stand for and-connectives with different procedural semantics (Epilog in [Por82]), or by a labelled tree where labels count how much consecutive resolution steps may be made in the same sub-goal-statement to achieve a fair strategy [Del86].

As with standard strategy, sharings are easy and secure since MALI is only involved with or-control.

3.2 Or-control

In depth-first search strategy, or-control amounts to backtrack point creation and deletion, and backtracking. The corresponding commands of MALI must be used.

Interpretation of Prolog's cut requires to fetch the *Name* of the most recent *Level* (`CurLevel`) before entering a clause where a cut occurs, and to store it until execution of the cut. This *Level* represents a state of the search stack. Executing the cut amounts to execute the command `Cut` with the stored *Name* as argument.

Prolog could inherit from the greater generality of MALI, and explicitly offer a cut with argument. In this way, the '!' would be a notation for the state of the search stack before entering the clause where it occurs. It could be passed as parameter to goals, substituted to variable and be the parameter of the predicate that cuts.

Non-depth-first search cannot be so easily mapped on MALI's structure. It can be coded on *Terms* but shall not benefit from as much efficient memory management as it could be. In the same way as depth-first search (e.g. Prolog) usefulness logic is different from no-search (e.g. Lisp) usefulness logic, non-depth-first search has its peculiarities. As we do not know of an efficient algorithm to implement its usefulness logic, there is no provision for this in MALI.

3.3 Unification

3.3.1 First-order terms unification

The first unification we consider is the first-order terms unification. It is straightforward in MALI. It amounts to recursively traverse one (static unification) or two (dynamic unification) *Terms*, and to test their *Nature* and *Sort*. However, to be efficient, our actual procedures heavily use MALI's advanced commands.

3.3.2 Type unification

Type unification (see LOGIN [AN86]) is a unification on an extension of terms which yields inheritance, not only through substitution of variable, but also through substitution of functional symbol, and increasing of arity. The 'signature' is a partial order on type symbols which is extended to typed terms built upon the type symbols. Two special types are the type of all objects and the type of no object.

A typed term is represented by an *Attributed Variable* whose *Attribute* is a pair formed of a type symbol (an *Atom*) and a consed list of labelled arguments. A labelled argument is a pair formed of a label (an *Atom*) and a typed term.

Unifying two typed terms amounts to find the greatest lower type of the two types according to the partial order on types. If it is the type of no object, unification fails. Otherwise, the two lists of arguments are merged label-wise. Equally labelled arguments are recursively unified, and the *Attributed Variables* representing the two terms are bound to a new *Attributed Variable* that represents a typed term formed of the greatest lower type symbol and the merged and unified arguments.

Through unification, each typed term inherits from the arguments of the other. After several type unifications, long chains of bound *Attributed Variables* may appear. The case is more serious than for simple *Variables*, because their *Attributes* make *Attributed Variables* consume unbounded amount of memory. The variable-shunting capability of the collector insures that *Attributed Variables* that represent bound typed terms will disappear as soon as possible.

3.4 'Freeze' and 'Dif'

As a special case of non-standard Prolog resolution, we show how we use MALI to implement the predicates 'Freeze' and 'Dif' of PrologII [Van84].

'Freeze' allows to delay the resolution of a goal until some variable is bound to a non-variable term. The variable is called the trigger, and the goal the frozen goal.

'Dif' is a sound implementation of the negation of equality. Though very different, it uses the same interpretation scheme as 'Freeze'. The semantics of a successful 'Dif' is that there exists a substitution that makes its arguments different. If the arguments are not unifiable, then 'Dif' succeeds; if they are formally equal, then it fails. In other cases, it is frozen with at least one trigger which is a free variable of one of the argument. Another difference with 'Freeze' is that a frozen 'Dif' only waits for the trigger to be bound, even to a variable.

3.4.1 First solution: the fridge

Our first solution [LR86] is a very versatile solution, but expensive.

Frozen goals are put in a list, the 'fridge', where they wait the binding that will unfreeze them. This is an associative list where frozen goals are paired with their trigger. When a trigger is bound, the associated goals are searched, picked out of the list and prepended to the goal statement.

To recognise the freezing variables, two *Sorts* of *Variable* are used to represent freezing and non-freezing variables. When a non-freezing variable becomes freezing, it is bound to a variable whose *Sort* is 'freezing'.

The fridge is a companion data structure to the goal statement and both must be saved, retrieved and reduced at the same time. The unfrozen goals are nicely discarded, and do not pollute further computation. It is not true in the standard implementation where a trigger keeps track of all the goals that have been frozen waiting upon its binding.

It is a versatile solution because it allows for many extensions: triggering with more than one variable, triggering on any monotonic condition (not only occurrence of binding), and easy control of the set of frozen goals. Several fridges could be used to implement a priority policy.

But the associative search of triggered goal makes this method expensive.

3.4.2 Second solution: the *Attributed Variable*

Our second solution is less versatile, but offers a direct access to triggered goals.

When a goal has to be frozen, the corresponding variable is bound to an *Attributed Variable* whose *Sort* is 'freezing' and *Attribute* is the frozen goal. Then, the binding of a trigger consists in getting the *Attribute*, putting it in the goal statement and binding the *Attributed Variable*.

Note that these actions are easy since frozen goals and goal statements have a similar representation.

For both solutions, unfreezing occurs during unification. In case of unfreezing, the semantical differences between 'Freeze' and 'Dif' are of importance. 'Dif' is part of unification (i.e. evaluation and updating of constraints), though 'Freeze' is only a strategy for selecting goals. So unfrozen 'Difs' must be evaluated before other unfrozen goals.

4 MALI's programming style is Prolog's

The programming styles in MALI and Prolog are very similar in several respects. In both cases the programmer is not concerned with memory management (this assertion is true only if Prolog is supplied with a garbage collector). He does not have to program the garbage collector of his application within the application. The data structures are very comparable; terms with variables. The control structures are also similar; the non-determinism is managed automatically either by MALI or Prolog. The user is unaware of whether a modification in the data structures will be backtracked or not. So, he does not have to explicitly trail such a modification. The state of Prolog is the set of all active goal statements. It corresponds to the set of saved *Terms*. MALI has *Attributed Variables*, but Prolog could be supplied with, either explicitly or implicitly through an extension of Prolog (e.g. LOGIN).

Having an efficient memory management favours a more declarative programming style because the user has not to think in terms of low level grounds (e.g. memory allocation). Then, the 'assert' and 'retract' predicates lose one of their most frequent use: memory management by asserting the computation state, then backtracking to recover memory, and then fetching the asserted state and retracting it. Since garbage collection is done automatically and is complete, there is no need to use these predicates to do memory management. The semantics of these predicates can be restricted to fit more accurately with logic (say for example, modal logic). It is well known that these predicates are logically unsound. We believe they are algorithmically troublesome too. They enforce the user to represent data (for instance, graphs or grammars) as predicates instead of as terms, thus replacing structural properties of terms by computational properties of predicates.

Less happily, the programming difficulties are almost the same in the two languages. Prolog restrictions on assignment forbid to write some efficient algorithms in Prolog. For example, efficient unification algorithms use assignment to gain qualitative advantages (complexity order or even termination). Some assignment modifies the representation of terms during unification; some other restores the representation after unification. This is impossible with Prolog or basic MALI, because the only way to undo something is to backtrack. To write these algorithms in MALI or Prolog, one has to substitute association list to assignment. To avoid it, we use the advanced commands of MALI.

5 Comparisons and conclusions

5.1 Comparison with WAM

MALI is a new concept in logic programming implementation studies, since it gives back the precedence to the declarative semantics rather than the procedural ones. Another abstract machine, WAM, is tightly bound to the procedural semantics of Prolog with a speed efficiency purpose.

It must be said that a design based on MALI may be not as time efficient as WAM in the case of strictly procedural semantics. This comes from the fact that goal statements must be represented in such a way that the collector can work. This may prevent compiling clause bodies as efficiently as with WAM. We are currently analysing this problem. This comes also from the greater generality of MALI, which is unemployed in the context of classical Prolog.

There is no such problem with the compilation of unification; it can be done in the same condition as WAM, with the advantage that all provision for backtracking is done in the machine.

The advantage of WAM becomes a disadvantage when non-procedural behaviour is introduced in logic programming languages. Non-SLD strategies, coroutines, lazy functional unification cannot be mapped on a procedural behaviour. The procedure-oriented features of WAM become inapplicable, though these new non-procedural behaviour seems to be part of the future of logic programming.

5.2 Comparison with the standard implementation of Prolog

In order to experiment MALI, we have designed a complete PrologII system whose resolution state is represented in MALI [Le 88].

Some aspects of PrologII ('Freeze' and 'Dif') have shown up the interest of MALI. The fact that the goal-statements are coded onto *Terms* makes easier every control-oriented operation. The independence between memory management and and-control has automatically solved the problem of the unfrozen goals.

On a PC/AT, and a software implementation of MALI, PrologII/MALI is four times as fast as regular PrologII. This shows that our fundamental technical choices do not impede speed efficiency. These choices are: (1) automatic memory management, (2) copy rather than static data structure-sharing, (3) the use of a unique compact memory space, instead of several specialised stacks.

The qualitative effect of MALI's memory management is obvious. Though very few Prolog systems can execute the 'naive reverse' program with a list of more than one hundred elements (this program with a list of thirty elements belongs to the standard Prolog benchmark), PrologII/MALI reverse more than 2500 elements with 64000 words of 32 bits.

System	Arity/Prolog	PrologII/MALI	PrologII
core memory	160K	240K	432K
max. num. of tail recursive calls	∞	∞	1700
max. num. of non-tail recursive calls	2703	14315	1700
largest representable int. list	4050	14000	1500
time to reverse a 100 elem. list	10"	9,7"	$\perp$
time to reverse a 500 elem. list	8'29"	4'6"	$\perp$
time to reverse a 900 elem. list	95'15"	17'16"	$\perp$
largest naively reversible int. list	900	2500	60
largest quick sortable int. list (already ordered)	85	> 2000	50

Table 1: Comparative measurements

We sum up in Table 1 measurements that have been done with Arity/Prolog, regular PrologII and PrologII/MALI. We have chosen Arity/Prolog because it has a garbage collector and it works on a PC. All are interpreters and the measurements have been done on an IBM PC/AT. The signs '∞' and '$\perp$' stand for 'no upper bound' and 'stopped with memory exhaustion'.

The Arity/Prolog system is equipped with a garbage collector that we do not know except that it seems to use a Morris [Mor78] compacting algorithm. The garbage collector of MALI has been

implemented with various garbage collecting techniques. The one of the comparison uses a copying algorithm [Che70]. Moreover, Arity/Prolog and PrologII/MALI differ by the usefulness logic that their respective memory managements implement. The first system uses a Lisp-oriented usefulness logic, and PrologII/MALI inherits from MALI a Prolog-oriented one. The standard PrologII system has no garbage collector. Its only memory recovering capability is to pop the search stack when backtracking. So, the comparison between the two implementations of PrologII shows the qualitative effect of an efficient memory management, and the comparison between Arity/Prolog and PrologII/MALI shows the effect of a more specialised usefulness logic.

The data giving the time to reverse a list of increasing length shows the effect of the garbage collecting technique. They show the progressive collapsing (analogous to system thrashing) of Arity/Prolog when the length of the list to be reversed increases. Collapsing is due to more and more frequent calls to the garbage collector whose execution time is proportional to the size of the whole memory, though PrologII/MALI's garbage collector execution time is proportional to the size of the useful memory.

The fact that there is no progressive collapsing on the last item, shows that the usefulness logic of Arity/Prolog is defective. Though there is room for garbage collecting, the system does not find it and stops suddenly.

When using MALI, there is a close relationship between execution optimisation and memory management. The memory management improvement comes from the fact that most speed optimisations avoid an excessive use of MALI. For instance, the effect of clause indexing is to suppress useless computation and also to avoid useless backtrack points, and then to retain less memory. We have introduced clause indexing in both interpreted and compiled versions of PrologII/MALI, because this feature must not be a privilege of compiled systems. Several experiments have been done in using MALI as a target machine for a PrologII compiler. One is a transposition of the WAM [Amr88]. The execution speed has been multiplied by six, and an improvement have also been observed in memory management, thus leading to a larger speed ratio for big data structures. The most accelerating features was clause indexing and special treatment of first goals of clause bodies. Other experiments are a compiler which performs data-flow optimisation and partial evaluation [Lan88] and, in connection with the software engineering society 'CRIL', a compiler for a dialect of Prolog called Prolog/P.

5.3 Conclusions

MALI is a new concept for the implementation of logic programming, for it gives back the precedence to the logical semantics rather than the procedural ones. Since it offers a rather high level interface (the *Term*), a reliable memory management (whichever and-strategy is adopted), and includes the depth-first strategy for or-control, MALI is easily used to implement logic programming languages, standard Prolog or not.

It remains an important point to explore: the extension of MALI to the non-depth-first search strategy. We have shown that it cannot be as time efficient as in depth-first case. It must perform a garbage collection for each active or-process. We are studying how to make these multiple garbage collections share as much work as possible. This extension of the semantics of MALI will provide a general tool for implementing logic programming languages.

References

[Amr88] M. Amraoui. *Une expérience de compilation de PrologII sur MALI.* Thèse, Université de Rennes I, 1988.

[AN86] H. Ait-Kaci and R. Nasr. Login: a logic programming language with built-in inheritance. *Journal of Logic Programming*, (3):185–215, 1986.

[BCRU84a] Y. Bekkers, B. Canet, O. Ridoux, and L. Ungaro. A memory management machine for Prolog interpreters. In Sten-Ake Tarnlund, editor, *2nd Int. Logic Programming Conference*, pages 343–351, Uppsala University, 1984.

[BCRU84b] Y. Bekkers, B. Canet, O. Ridoux, and L. Ungaro. *Spécification d'une machine de gestion de mémoire pour les interpréteurs des langages logiques — Version 1.* Publication Interne 222, IRISA, 1984.

[BCRU86] 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 Symposium on Logic Programming*, IEEE, 1986.

[Che70] C.J. Cheney. A nonrecursive list compacting algorithm. *CACM*, 13(11):677–678, 1970.

[Del86] J.P. Delahaye. Définitions de stratégies équitables en programmation logique. In *Séminaire de Programmation Logique de Trégastel*, CNET, France, 1986.

[Lan88] G. Landais. *Compilation de Prolog par évaluation partielle.* Thèse, Université de Rennes 1, 1988.

[Le 88] S. Le Huitouze. *Mise en œuvre de PrologII/MALI.* Thèse, Université de Rennes 1, 1988.

[LR86] S. Le Huitouze and O. Ridoux. Une expérience de réalisation du Gel et du Dif dans MALI. In *Séminaire de Programmation Logique de Trégastel*, CNET, France, 1986.

[Mor78] F.L. Morris. A time and space efficient garbage compaction algorithm. *CACM*, 21(8):662–665, 1978.

[PBW85] E. Pittomvils, M. Bruynooghe, and Y.D. Willems. Towards a real time garbage collector for Prolog. In *2nd Symposium on Logic Programming*, IEEE, 1985.

[Por82] A. Porto. Epilog: a language for extended programming in logic. In *1st International Logic Programming Conference*, 1982.

[Rid86] O. Ridoux. *Gestion de mémoire temps-réel des langages de programmation relationnelle.* Thèse, Université de Rennes I, 1986.

[Rob65] J.A. Robinson. A machine-oriented logic based on the resolution principle. *JACM*, 12(1), 1965.

[Van82] M.H. Van Emden. An interpreting algorithm for Prolog programs. In *1st International Logic Programming Conference*, 1982.

[Van84] M. Van Caneghem. *L'anatomie de PrologII.* Thèse de doctorat d'état, Université d'Aix-Marseille, 1984.

[War77] D.H.D. Warren. *Implementing Prolog — Compiling Logic Programs, Vol. 1 and 2.* D.A.I. Research Report 39, 40, University of Edinburgh, 1977.

[War80] D.H.D. Warren. An improved Prolog implementation which optimises tail-recursion. In *Logic Programming Workshop*, Debrecen, Hungary, 1980.

[War82] D.H.D. Warren. Perpetual processes — an unexploited Prolog technique. *Logic Programming Newsletter*, (3), 1982.

[War83] D.H.D. Warren. *An Abstract Prolog Instruction Set.* Technical Note 309, SRI International, 1983.

LISTE DES DERNIERES PUBLICATIONS INTERNES

PI 463 **ELEMENTS FINIS C^1, POLYNOMIAUX DE DEGRE QUATRE PAR TRIANGLE, DANS UNE TRIANGULATION FORMEE DE TRIANGLES EQUILATERAUX**
Michel CROUZEIX, Miloud SADKANE
26 Pages, Mars 1989.

PI 464 **VERS UN MODELE D'ECLAIREMENT REALISTE**
Pierre TELLIER, Kadi BOUATOUCH
66 Pages, Avril 1989.

PI 465 **COMPILATION OF FUNCTIONAL LANGUAGES BY PROGRAM TRANSFORMATION**
Pascal FRADET, Daniel LE METAYER
28 Pages, Avril 1989.

PI 466 **MODEL BASED DIAGNOSIS : A CASE STUDY IN VIBRATION MECHANICS**
Michèle BASSEVILLE, Albert BENVENISTE
26 Pages, Avril 1989.

PI 467 **DELAI DE COMMUNICATION ENTRE NOEUDS VOISINS SUR L'iPSC/2**
Patrice BURGEVIN, André COUVERT, René PEDRONO
16 Pages, Avril 1989.

PI 468 **OBSERVING GLOBAL STATES OF ASYNCHRONOUS DISTRIBUTED APPLICATIONS**
Jean-Michel HELARY
24 Pages, Avril 1989.

PI 469 **ON-LINE MODEL CHECKING FOR FINITE LINEAR TEMPORAL LOGIC SPECIFICATIONS**
Claude JARD, Thierry JERON
16 Pages, Mai 1989.

PI 470 **PROGRAMMING WITH MALI - THE INTERPRETATION OF PROLOG PROGRAMS**
Louis CHEVALLIER, Serge LE HUITOUZE, Olivier RIDOUX
16 Pages, Mai 1989.

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique