Programming with MALI -unification or ordered types

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° 1058

Programme 1

PROGRAMMING WITH MALI - UNIFICATION OR ORDERED TYPES

Olivier RIDOUX

Juillet 1989

RR-1058

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° 477 - Juin 1989 - 18 Pages

Programming with MALI — Unification or Ordered Types

Olivier Ridoux

Abstract

The introduction of ordered types and the associated unification in logic programming gives a mean to express an 'isa' taxonomy and to integrate an inheritance calculus. A two phases method is proposed for the implementation of the unification of ordered types. The first phase is executed once, at compile time. It performs the coding of taxonomic symbols in such a way that the computation of the greatest lower bound of two symbols in the 'isa' lattice is merely a bitwise 'and' of the codes. The second phase performs unification. It uses 'MALI', a logic programming oriented memory managing machine, and specially one of its feature, called 'attributed variable'. Thus, the proposed unification procedure is both time and space efficient.

Programmer avec MALI — l'unification des types ordonnés

Résumé

L'introduction des types ordonnés et de l'unification associée dans la programmation logique permet d'exprimer une taxinomie d'appartenance et d'intégrer un calcul d'héritage. Une méthode à deux phases est proposée pour l'unification des types ordonnés. La première phase est exécutée une fois, à la compilation. Elle réalise le codage des symboles taxinomiques de telle manière que le calcul de la borne inférieure de deux symboles du treillis d'appartenance soit simplement un 'et' bit à bit des codes. La seconde phase réalise l'unification. Elle utilise 'MALI', une machine de gestion de mémoire pour la programmation logique, et particulièrement un de ses dispositifs appelé 'variable à attribut'. Ainsi, la procédure d'unification proposée est efficace à la fois en temps et en espace.

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

In [AN86], Aït-Kaci and Nasr present an extension of Prolog, called 'LOGIN', enhancing the power of the language for expressing inheritance among objects described by an 'isa' taxonomy. They advocate that, though the standard first order terms have the theoretical capability to encode such a calculus, they do not fit this purpose. A first order term can be interpreted as a set (for instance, the set of all its closed instances) and unification can be interpreted as set intersection. In this way, binding a variable achieves the effect of inheritance. Two limitations of first order terms make this method pragmatically unconvenient. First, variables being the only medium for inheritance, the programmer must provide terms with free variables as slots to receive inherited values. Second, functional symbols arity being fixed, the slots must be ready since the beginning of the computation. Furthermore, variables play two distinct roles: the greatest class of the taxonomy and equality constraints among the multiple occurrences of the same variable.

LOGIN extends Prolog by relaxing the two previous limitations. Functional symbols become the medium for inheritance. They are partially ordered by a 'signature' given by the programmer. This signature must be a lower semilattice. The greatest functional symbol plays the first role formerly given to variables, and tags, which are allowed to occur everywhere in a term, play the second role. Arity is no longer fixed and terms arguments are explicitly labelled rather than being positionally referenced. The partial ordering of symbols is extended to a partial ordering of terms and unification amounts to a greatest lower bound (GLB or meet) operation.

MALI (Machine Adaptée aux Langages Indéterministes — machine suited to non-deterministic languages) [BCRU86, CLR87] is a logic programming oriented machine whose specification admits an efficient garbage collecting implementation. The idea is that the common feature of logic programming languages —non-determinism— must be taken into account to have an efficient memory management. In order to handle non-determinism, MALI must know about logical variables and the management of the search-space. Since MALI is bound to depth-first search, it knows about state saving and retrieving. In this way, the logic programming language implementor is relieved from the burden of the memory management and non-determinism as when a programmer uses Prolog.

We present an efficient implementation of LOGIN on MALI. It shows the practicality of LOGIN and the generality of MALI. All these points are detailed in three parts. In the first two parts, the principles of LOGIN and the relevant features of MALI are presented. In the third part, we present our implementation of ordered terms unification. A basic knowledge of finite lattice theory is assumed (a good reading can be [RW88]).

2 LOGIN

Unification can be considered as the computation of the greatest lower bound of two terms in a lower semilattice [Hue76]. This is a more general point of view than the classical least substitution computation. Standard first order unification is defined by the following partial order among first order terms:

$$t_1 \leq t_2 \equiv (\exists \sigma)(\sigma t_2 = t_1).$$

This partial order is called the 'subsumption' relation. LOGIN's 'type unification' or 'ψ-term unification' is also defined by a lower semilattice.

We quote Aït-Kaci and Nasr to define ψ-terms and a partial order on them.

> *Informally, a ψ-term consists of:*
>
> 1. *A 'root symbol', which is a type constructor and denotes a class of objects.*
> 2. *'Attribute labels', which are record field symbols, associated with ψ-terms. Each label denotes a function 'in intenso' from the root type to the type denoted by its associated sub-ψ-term. Concatenation of labels denotes function composition.*

3. *'Coreference' constraints among paths of labels, which indicate that the corresponding attribute compositions denote the same functions. In other words, coreference specifies that some functional diagram of attributes must be commutative.*

```
person (name   => X : string ;
        father => person (name => X))
```

Figure 1: a ψ-term

Coreference constraints are denoted by 'tags'. All ψ-terms with the same tag must be equal. We give in Figure 1 a sample ψ-term in the syntax of LOGIN. The root symbols are **person** and **string**, the attribute labels are **name** and **father** (=> points to the ψ-term associated to the attribute label at its left), and there is one coreference tag which is **X**. In a type as set interpretation, **person** and **string** denote sets and this ψ-term can be paraphrased in "the sub-set of the persons whose name is the same as their father's". In a real-life implementation, **string** can be a predefined type. In the following, we suppose that every ψ-term is tagged, thus every ψ-term is identified by a tag.

The set S of type constructors is called the 'signature' and is supposed to be partially ordered by type inclusion ($\sqsubset$). We note Σ the structure $(S, \sqsubset)$: it must be a lower semilattice. It always contains a greatest element '$\top$', which denotes the 'universe', and a least element '$\bot$', which is the type of no object. The partial order on the signature is extended to the ψ-terms in the following way:

Informally, a ψ-term t_1 is a 'subtype' of a ψ-term t_2 if:

1. *the root symbol of t_1 is a subtype in Σ of the root symbol of t_2;*
2. *all attribute labels of t_2 are also attribute labels of t_1, and their ψ-terms in t_1 are subtypes of their corresponding ψ-terms in t_2; and*
3. *all coreference constraints binding in t_2 must also be binding in t_1.*

If symbols in $\Sigma - \{\top, \bot\}$ are incomparable, and if for all terms, the set of their attribute labels is an integer interval with the same lower bound, and if tags only occur on leaves with $\top$ as associated type, then ψ-terms are first order terms.

In order to describe a unification on ψ-terms, we will translate the notion of substitution to take advantage from its operational counterpart. We define a substitution in the ψ-terms as an almost identity decreasing mapping. The standard definitions (composition, least substitution) can be translated as well. Informally, the substitution `S` which yields the GLB of two ψ-terms `T1` and `T2` is computed by the algorithm of Figure 2. Its local variables are `S` of type "substitution" and `PSI_EQ` of type "set of pairs of ψ-terms". `PSI_EQ` stores the pairs of sub-terms of `T1` and `T2` that remain to be unified.

This naive unification algorithm is not meant to be efficient. Its only purpose is to point out two difficulties. They are the computation of GLB in Σ and the representation of substitution effects. The algorithm presented in [AN86] is more elaborated than ours in that it describes a representation of the substitution effects but it does not solve these difficulties either. In both algorithms, the first problem is hidden in the GLB operation and the second in the 'union' operation of our algorithm, or in the use of sets as data structure to represent attribute labels in the other algorithm. The problem with the GLB is that it is a key operation, most frequently used, for which a graph-like implementation is too inefficient. Assuming that Σ is represented as a graph, computing the GLB requires associative search in the graph. The problem with substitution is that each binding value (`Z` in Figure 2) requires new memory resources for its representation. At a given time, a ψ-term is represented by a chain of

```
UNIFY (T1,T2) -> {(UNIFIABLE,substitution), NOT UNIFIABLE}
begin
  S <- identity;   PSI_EQ <- {(T1,T2)};
  while PSI_EQ <> {} do
    let (X,Y) in PSI_EQ
    PSI_EQ <- PSI_EQ - (X,Y);
    let SX = S(X)
    let SY = S(Y)
    if SX <> SY then
      let RZ = the GLB in SIGMA of ROOT SYMBOL of SX and ROOT SYMBOL of SY
      if RZ = BOTTOM then return ( NOT UNIFIABLE )
      else
        let ASX = ATTRIBUTE LABELS of SX
        let ASY = ATTRIBUTE LABELS of SY
        let Z = the PSI_TERM whose
            ROOT SYMBOL is RZ,
            ATTRIBUTE LABELS are ASX union ASY
            and COREFERENCE CONSTRAINTS are the union of those of SX and SY
        S <- (SX/Z) o (SY/Z) o S;
        for each AZ in ASX inter ASY do
          let Z1 = the PSI_TERM ASSOCIATED to AZ in SX
          let Z2 = the PSI_TERM ASSOCIATED to AZ in SY
          PSI_EQ <- PSI_EQ union {(Z1,Z2)};
        od;
      fi;
    fi;
  od;
return ( UNIFIABLE, S );
end
```

Figure 2: a unification algorithm

bindings, among which only the last binding value and those that are saved for backtracking need to be represented. This problem arises also with standard Prolog terms, since a bound variable whose free state is not saved for backtracking need not be represented as a variable nor trailed. Its binding value can be assigned to all its occurrences. It clearly appears in the case of binding chains but is not as crucial as for ψ-term, because variables are represented by (small) constant size objects though ψ-terms require non-constant, unbounded size objects. In short, all the 'avatars' of a ψ-term need not be represented. Only the last one and those that can be uncovered by backtracking must be.

3 MALI

The study of Prolog implementation is generally oriented towards computing speed. However, this propensity leaves a problem unsolved: memory management. MALI is a logic programming oriented intermediate machine that results from the opposite inclination. MALI has shown to be really logic programming oriented and not only Prolog oriented. Its only involution with a particular implementation choice is with the depth-first strategy to handle non-determinism. It is not at all involved with and-control or with the underlying term theory. So MALI implements a memory management theory that is only specific to depth-first search though it is more precise than the Lisp-like theory. It transparently handles allocation and collection of memory resources.

The specification of MALI defines a memory and its access protocol. The protocol must indeed be compatible with the memory management theory but this still leaves room for specification choice. MALI's choice is to ease comparison and transfer between MALI's data and non-MALI's data. It seems that most logic programming implementations require efficient comparison and transfer between the program (non-MALI's data) and the evaluation state (MALI's data).

There are two implementations of MALI.

1. A IBM-PC/XT/AT compatible hardware implementation with a real-time [Bak78] garbage collector. The garbage collector works as a background task and is interrupted when primitives ('commands') of the protocol are executed.

2. A software implementation written in C. Several implementations of the garbage collector have been realised. One of them is real-time, the others, in which the garbage collector is triggered near exhaustion time, are not.

We now present the concepts of MALI that are relevant for ψ-term unification. 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 embedded in MALI and is not the responsibility of the programmer using MALI to implement a logic programming system, thus simplifying his work. Furthermore, there is a term construction and traversal capability that provides the kernel of value representation for the user.

In the following, identifiers of MALI's specific concepts are written with an upper case initial. The commands of MALI have arguments of a specific type called 'Names', and occasionally integers and booleans. The state of MALI is composed of a partial function, the 'Designation', which associates a 'Term' to a Name, and a stack of Terms, the 'Search Stack'. So, Name and Designation are the MALI correspondents to random memory address and contents. Names contain a tag that indicates the 'Nature' and 'Sort' of a Term. In the scope of this paper, the Nature can be either 'Atom', 'Construct' or 'Attributed Variable'. The Sort gives elementary typing to Atom, Construct or Attributed Variable. We give the semantics of MALI through its commands. The profile of commands is noted

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

where the `Ii`'s (resp. `Oj`'s) are the input (resp. output) parameters types. Initials of Nature identifiers (A, C, AV) 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 returns the Nature and Sort of a named Term. The user knows the structure of an Atom Name, and can build bit patterns that look like Atom Names for representing its own atoms values. All other Names are synthesised by MALI itself.

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

The result of 'Cons' is a Name of the consed Term whose sub-Terms are the Terms that are designated by the two parameters. Access to sub-Terms is done with 'RightSubCons' and 'LeftSubCons'. All this is very similar to 'cons', 'car' and 'cdr' in Lisp.

The following commands define the use of Attributed Variables. As Variables they behave like the logical variables of Prolog.

```
AttrdVar                      (AV_Sort,Name) -> (AV_Name)
BindAttrdVar                  (AV_Name,Name) -> ()
VarAttr                       (AV_Name) -> (Name)
```

The command 'AttrdVar' returns a Name of a new Attributed Variable. The effect of 'BindAttrdVar' is to substitute a Term to an Attributed 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. Since MALI handles non-determinism, it automatically undoes substitution when backtracking. An Attributed Variable has as an associated value a Term called 'Attribute'. The command 'VarAttr' returns a Name of the Attribute of an Attributed Variable. Attributed Variable is a key features of MALI for the implementation of ψ-terms, but it has first been used for the implementation of the Dif and Freeze predicates of PrologII [Van84, Le 88]. In this application, Attributed Variables represent constrained or triggering PrologII variables, and their Attributes represent constraints or frozen goals.

We briefly present the features of MALI memory management. All its novelty comes from the fact that the evolution of the Designation is constrained by the depth-first strategy. So, the garbage collector can interpret the representation of Variables according to the Search Stack. This novelty is entirely accounted for in MALI.

For instance, the collector can detect that it is useless to represent the substitution of a Variable. This happens when the representation of a Variable is only reachable from the representation of backtrack points where it is not considered to be substituted. Symmetrically, the collector can detect that a Variable is definitively substituted and that it is useless to represent it and its Attribute. Such a Variable is considered substituted in every backtrack point from which it is reachable. When this happens, the collector replaces the representation of the Variable by the representation of the substitution value. These features (that we call 'binding-shunting' and 'variable-shunting') are needed to program a 'perpetual process' as in [War82]. They make the 'scaffolding' of term construction disappear as soon as they become useless. Variable-shunting is still more important with Attributed Variables, since they can be discarded with their Attribute. In classical implementations of Prolog, those scaffoldings remain and a perpetual process eventually exhausts the memory.

4 Unification of ordered terms

We solve the GLB operation problem by a special coding of symbols of Σ. This coding is such that the GLB operation amounts to a bitwise 'and'. It is performed at compilation time. The substitution problem is solved by the use of Attributed Variables to represent coreference tags.

4.1 Compilation time phase

The compilation time phase aims at coding symbols of Σ in a target lattice of binary words such that the inclusion relation is preserved. A lattice of n-bits words with bit-wise 'and' and 'or' is equivalent to the lattice of the subsets of an n-elements set (we note it 2^n) with intersection and union. There are strong constraints on the size of the target lattice. A first naive constraints is that the number of bits in the binary words must be sufficient to count the elements of Σ. A sample symbols lattice immediately shows that this constraint is too weak. We give an expression of the number of bits needed to code a given Σ and an algorithm to assign a code to each symbol. The theory of LOGIN only requires that Σ is a lower semilattice. Our coding requires that it is a lattice. We want that intersection and also union make sense in Σ. This does not restrict the applicability of our coding since a finite partial ordered set can always be completed in a powerset [Ait86].

Definition 1 *Given $\Sigma = (S, \sqsubset)$, R_Σ is the relation such that*

$$x \, R_\Sigma \, y \equiv x \sqsubset y \wedge \neg(\exists z \in S)(x \sqsubset z \sqsubset y).$$

This relation is the direct descendant relation of Σ and corresponds to its Hasse diagram.

Definition 2 *Given $\Sigma = (S, \sqsubset)$, $S_\cup$, $S_\cap$ and $S_\subset$ form a partition of S such that*

$$S_\cup = \{\bot\} \cup \{x \in S \mid (\exists y, z \in S)(y \neq z \wedge y \, R_\Sigma \, x \wedge z \, R_\Sigma \, x)\}$$

(These vertices denote set union),

$$S_\cap = \{x \in S - S_\cup \mid (\exists y, z \in S)(y \neq x \wedge z \, R_\Sigma \, x \wedge z \, R_\Sigma \, y)\}$$

(These vertices are related by set intersection),

$$S_\subset = S - (S_\cup \cup S_\cap)$$

(These vertices denote set inclusion).

The elements of $S_\cap$ and $S_\subset$ are the join-irreducible elements of Σ. It is a well-known result that every element of a finite lattice can be written as a join of join-irreducible elements.

Definition 3 *Given $\Sigma = (S, \sqsubset)$ and an integer n, a strict set-coding function is a function $f : S \to 2^n$ such that*

$$(\forall x, y \in S)(x \neq y \Rightarrow f(x) \neq f(y)), \tag{1}$$

$$f(\bot) = \emptyset, \tag{2}$$

$$(\forall x, y \in S)(f(GLB(x, y)) = f(x) \cap f(y)). \tag{3}$$

Condition 3 of definition 3 enforces the equivalence between GLB computation and bitwise 'and'. There is not a strict set-coding function for all Σ and n. For instance, if n is 0 and S is not a singleton, condition 1 cannot be satisfied. We give a sufficient and necessary condition on n to have a strict set-coding function.

Lemma 1 *Given $\Sigma = (S, \sqsubset)$, $C : S \to 2^{\| S_\cap \cup S_\subset \|}$, such that $C(x) = \{y \in S_\cap \cup S_\subset \mid y \sqsubseteq x\}$, is a strict set-coding function.*
($\|x\|$ reads 'number of elements of x').
Proof: *To show that C satisfies the three conditions of definition 3.*

1

Assume x and y are two different join-irreducible elements ($\in S_\cap \cup S_C$),

then x or y is different from $GLB(x,y)$, assume x is,

then $x \in C(x) \wedge x \notin C(y)$, then $C(x) \neq C(y)$.

Any different x and y can be written differently as the join of join-irreducible elements,

their encodings are the unions of the encodings of their lower join-irreducible elements,

then their encodings are different.

2

Since $\perp \notin S_\cap \cup S_C$ then $C(\perp) = \emptyset$.

3

$$\begin{aligned} C(GLB(x,y)) &= \{z \in S_\cap \cup S_C \mid z \sqsubseteq GLB(x,y)\} \\ &= \{z \in S_\cap \cup S_C \mid z \sqsubseteq x \wedge z \sqsubseteq y\} \\ &= C(x) \cap C(y). \end{aligned}$$

□

Lemma 2 *Given $\Sigma = (S, \sqsubset)$ and an integer n, if a function C' maps Σ to 2^n and $n < \| S_\cap \cup S_C \|$ and C' satisfies conditions 2 and 3 of definition 3 then C' cannot satisfy condition 1.*
Proof: *Let C be the strict set-coding function defined in lemma 1.*

Then $(\exists x)(\|C'(x)\| < \|C(x)\|)$.

Let T be the set of all x for which $\|C'(x)\| < \|C(x)\|$ holds.

According to condition 2, $\|C(\perp)\| = 0$, and then $\perp \notin T$.

Let t be a minimal element of T.

A consequence of condition 3 is

$$\bigcup_{y \sqsubset x} C'(y) \subseteq C'(x).$$

Since t is minimal, it cannot be in $S_\cup$, because at least one of its descendant would be in T.

Then t is in $S_C \cup S_\cap$, and it has one direct descendant s,

then $\|C(t)\| = \|C(s)\| + 1 > \|C'(t)\| \geq \|C'(s)\| = \|C(s)\|$,

then $C'(t) = C'(s)$. Which violates condition 1.

□

Theorem 3 *Given $\Sigma = (S, \sqsubset)$ and an integer n, 2^n can code Σ with a strict set-coding function if and only if $n \geq \| S_\sqcap \cup S_\subset \|$.*
Proof:

$\Leftarrow$ *To prove that if $n = \| S_\sqcap \cup S_\subset \|$ then 2^n can code Σ follows from lemma 1.*

$\Rightarrow$ *To prove that if 2^n can code Σ then $n \geq \| S_\sqcap \cup S_\subset \|$ follows from lemma 2.*

□

An algorithm to compute a strict set-coding function uses the definition of C in lemma 1. C can be constructed level-wise from $\bot$. Then every element of $S_\sqcap \cup S_\subset$ is numbered. Binary words are deduced where each binary position is assigned a '1' if the corresponding element is present and a '0' otherwise. Since this algorithm is not incremental, it requires the assumption that no new symbol can be introduced at execution time (through queries or inputs).

The number of bits ($\| S_\sqcap \cup S_\subset \|$) required to code a signature can impede the application of the system. However, condition 1 of definition 3 can be relaxed if we are not interested in the solution substitution of a LOGIN program, but only in the existence of a solution. One can, for instance, remove the set $S_\subset$ from the code. This means to mingle each element of $S_\subset$ with its immediate descendant. We think that a more radical improvement must be made. For this purpose, the following is the definition of a more compact coding.

Definition 4 *Given $\Sigma = (S, \sqsubset)$ and an integer n, a sparing set-coding function is a function $f : S \rightarrow 2^n$ such that*

$$(\forall x \in S)(x \neq \bot \Rightarrow f(x) \neq f(\bot)) \tag{4}$$

and conditions 2 and 3 of definition 3.

With a sparing set-coding function, the elements of S are coded on the atoms of Σ instead of on its join-irreducible elements. It is a well known result that not all lattices can have their elements expressed as the join of their atoms. Boolean lattices can, but the lattice of Figure 3 cannot.

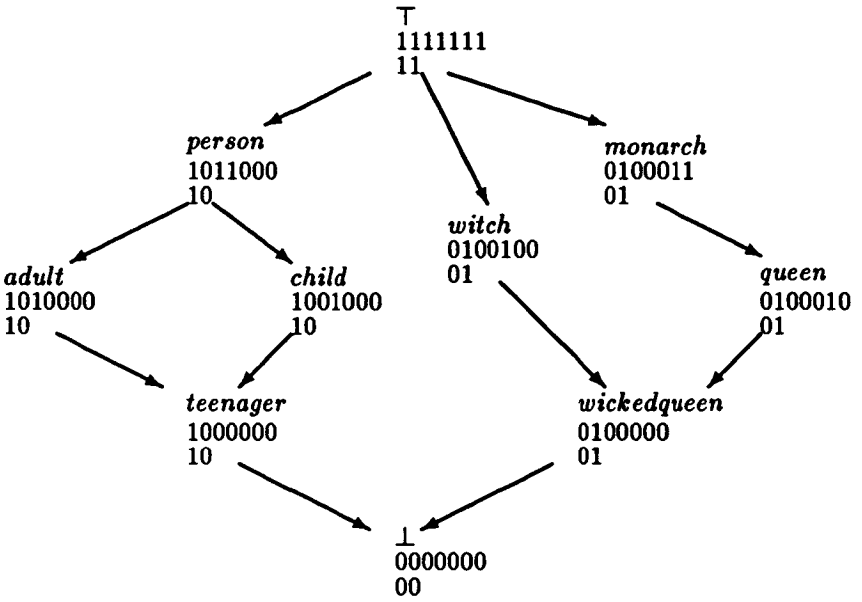
Theorem 4 *Given $\Sigma = (S, \sqsubset)$ and an integer n, 2^n can code Σ with a sparing set-coding function if and only if $n \geq \|\{x \in S \mid \bot R_\Sigma x\}\|$.*
Proof: *Let $C'' : S \rightarrow 2^{\|\{x \in S \mid \bot R_\Sigma x\}\|}$, such that $C''(x) = \{y \in S \mid y \sqsubseteq x \wedge \bot R_\Sigma y\}$.*

That it is a sparing set-coding function follows directly from definition 4.

That no sparing set-coding function uses less bits follows from conditions 2 and 3 of definition 3 and from the definition of C''.
If less than $\|\{x \in S \mid \bot R_\Sigma x\}\|$ bits are used,
then either two atoms have the same code, so the bitwise 'and' of their codes is not the code of their GLB (which is $\bot$),
or one atom has the same code as $\bot$, which violates condition 4 of definition 4.

□

With both set-coding functions, the worst case coding (largest number of bits required) is reached when every element of S is in $S_\sqcap \cup S_\subset$ or in $\{y \in S \mid \bot R_\Sigma y\}$. For instance, a flat (completely ordered —'vertical'— or completely unordered —'horizontal'—) lattice is a worst case for the strict set-coding function since $\|S\|$ bits are required. A flat horizontal lattice is also a worst case for the sparing set-coding function, but a flat vertical lattice is a best case for the same function since it requires only one bit. The best case for the strict set-coding function seems to occur when S is a



symbol
strict set-code
sparing set-code

Figure 3: an example of signature and its coding

boolean lattice: only $log_2||S||$ bits are required. In this case strict and sparing set-coding functions are identical. With both set-coding functions, we do not know the mean value for the number of required bits.

The coding and spare coding of a signature borrowed from [AN86] is shown in Figure 3. In this example,

$$\begin{aligned} S_{\mathsf{U}} &= \{person, \bot, \top\}, \\ S_{\mathsf{n}} &= \{adult, child, queen, teenager, wickedqueen, witch\}, \\ S_{\mathsf{C}} &= \{monarch\}. \end{aligned}$$

The binary positions of the strict set-coding function correspond to the join-irreducible root symbols in the following order: *teenager*, *wickedqueen*, *adult*, *child*, *witch*, *queen*, *monarch*. The binary positions of the sparing set-coding function correspond to the atomic root symbols in the following order: *teenager*, *wickedqueen*.

4.2 Unification time phase

We now assume that every root symbol has been coded into a binary word. The length of the binary word is supposed to be compatible with a MALI Atom. If it is not the case, sections of longer binary words can be appended with a MALI Term of Nature Construct. Moreover, attribute labels are totally ordered by arbitrary numbering and every ψ-term is identified by a coreference tag.

A ψ-term is represented by a MALI Term that imitates its structure.

1. The root symbol is represented by an Atom whose Sort is **#symbol** and value is the code of the symbol.

2. The attribute labels are represented by a list of Constructs whose Sort is **#label**. The list constructor and terminator are **#attribute** Sorted Constructs and Atom. The left sub-Term of a **#label** Construct is an Atom whose Sort is **#label** and value is the number of a label. Labels are enlisted in order of increasing number. The right sub-Term is the representation of the tag of the associated ψ-term.

3. The coreference tags are represented by Attributed Variables whose Sort is **#tag** and Attribute is a **#psi_term** sorted Construct. The left sub-Term of a **#psi_term** Construct is the representation of the root symbol of the tagged ψ-term. The right sub-Term is the representation of its attribute labels.

Figure 4 shows the representation of the ψ-term given in Figure 1. The second occurrence of `X` denotes the Term labeled by the first occurrence. The representation can be made more compact by using ternary constructors instead of Conses. But it would not enhance the readability of the unification procedures. According to the interpretation of the ψ-terms as types, an Attributed Variable represents a variable or a tag of LOGIN, and its Attribute represents its type. So, unification amounts to Attributed Variables substitutions, and Attributes comparisons.

Given this representation of ψ-terms, the procedures of Figure 5 and Figure 6 perform the unification of ψ-terms. The parameters of function **UNIFY** are the Names of two MALI's Terms representing tagged ψ-terms. The array **PSI_EQ** with **NB_PSI_EQ_IN** and **NB_PSI_EQ_OUT** is an operational version of its homonym of Figure 2. The main job of this function is to apply the bitwise 'and' operation (noted **&**) to the root symbols of two terms. The tag of one of the ψ-term is bound to the other which in turn is bound to a new ψ-term which is their GLB. This constructs the classes of equivalent ψ-terms in a way that is entirely transparent to the programmer. The function **MERGE_LABELS** returns the Name of a Term that represents the merged labels of two terms.

Dereferencing of coreference is done automatically by MALI since it specifies that the effect of BindAttrdVar is to replace every occurrence of the Attributed Variable by the Term. Reset information

```
AttrdVar(#tag, Cons(#psi_term, Atom(#symbol,person),
         Cons(#attribute, Cons(#label, Atom(#label,id),
          X : AttrdVar(#tag, Cons(#psi_term, Atom(#symbol,string),
                      Atom(#attribute,nil)))),
         Cons(#attribute, Cons(#label, Atom(#label,father),
              AttrdVar(#tag, Cons(#psi_term, Atom(#symbol,person),
                      Cons(#attribute, Cons(#label, Atom(#label,id),
                           X),
                      Atom(#attribute,nil))))),
         Atom(#attribute,nil))))
```

Figure 4: the representation of a ψ-term

```
{ Name, Name } PSI_EQ[];
int NB_PSI_EQ_IN, NB_PSI_EQ_OUT;

bool UNIFY( Name TAG1, TAG2 )
begin
  PSI_EQ[0] <- { TAG1, TAG2 };
  NB_PSI_EQ_IN <- 1;   NB_PSI_EQ_OUT <- 0;
  while (NB_PSI_EQ_OUT <> NB_PSI_EQ_IN) do
    let { TAG1, TAG2 } = PSI_EQ[NB_PSI_EQ_OUT]
    NB_PSI_EQ_OUT <- NB_PSI_EQ_OUT + 1;
    let ATTR1 = RightSubCons(VarAttr(TAG1))
    let ATTR2 = RightSubCons(VarAttr(TAG2))
    let GLB = LeftSubCons(VarAttr(TAG1)) & LeftSubCons(VarAttr(TAG2))
    if GLB <> 0 then
      BindAttrdVar(TAG1, TAG2);
      BindAttrdVar(TAG2, AttrdVar(#tag, Cons(#psi_term,
                                             Atom(#symbol,GLB),
                                             MERGE_LABELS(ATTR1,ATTR2))));
    else return ( false );
    fi;
  od;
  return ( true );
end/*UNIFY*/
```

Figure 5: the UNIFY procedure

```
Name MERGE_LABELS( Name ATTR1,   ATTR2)
begin
  while ( true ) do
    if NatureSort(ATTR1) = (Atom,#attribute) then return (ATTR2)
    elif NatureSort(ATTR1) = (Construct,#attribute) then
      if NatureSort(ATTR2) = (Atom,#attribute) then return (ATTR1)
      elif NatureSort(ATTR2) = (Construct,#attribute) then
        let NEXT_ATTR1 = RightSubCons(ATTR1)
        let LABEL1 = LeftSubCons(LeftSubCons(ATTR1))
        let TAG1 = RightSubCons(LeftSubCons(ATTR1))
        let NEXT_ATTR2 = RightSubCons(ATTR2)
        let LABEL2 = LeftSubCons(LeftSubCons(ATTR2))
        let TAG2 = RightSubCons(LeftSubCons(ATTR2))
        if (LABEL1 = LABEL2) then
          PSI_EQ[NB_PSI_EQ_IN] <- {TAG1,TAG2};
          NB_PSI_EQ_IN <- NB_PSI_EQ_IN + 1;
          return (Cons(#attribute,
                       Cons(#label, LABEL1, TAG1),
                       MERGE_LABELS(NEXT_ATTR1, NEXT_ATTR2)));
        elif LABEL1 < LABEL2 then
          return (Cons(#attribute,
                       Cons(#label, LABEL1, TAG1),
                       MERGE_LABELS(NEXT_ATTR1, ATTR2)));
        elif LABEL1 > LABEL2 then
          return (Cons(#attribute,
                       Cons(#label, LABEL2, TAG2),
                       MERGE_LABELS(ATTR2, NEXT_ATTR2)));
        fi;
      fi;
    fi;
  od;
end/*MERGE_LABELS*/
```

Figure 6: the MERGE_LABELS procedure

for backtracking is also handled by MALI. More complex commands allow to avoid the recursive construction of the representation of the attribute labels. These commands provide a facility to iteratively construct a Term according to a prefix Polish notation.

Useless avatars (neither the most recent, nor uncoverable by backtrack) are automatically garbage collected as an effect of variable-shunting applied to Attributed Variables.

5 Conclusion

A similar coding technique has been proposed independently by the authors of LOGIN [ABLN89]. They also propose some amendments of their coding in the taste of our sparing set-coding function (they collapse elements of S_C on their descendants instead of collapsing S on the direct antecedents of $\perp$). Noting, as we will do, that the coding can be fairly expensive, they also propose another coding where the GLB operation amounts to a conditional sequence of bitwise 'ands'. This more compact coding relies on the hypothesis that big taxonomies are defined in term of smaller ones that the authors call 'modules'. These authors do not describe the representation of ψ-terms and their bindings.

We have proposed a new implementation of LOGIN unification algorithm that is both space and time efficient. At least two problems remain.

The first problem is that the number of bits needed to code the root symbols can be fairly large. We do not know how large a real-life signature can be and how its Hasse diagram looks like. If the signature is large, it may be that it consists of several sub-signatures ('module') and that the belonging to one or another can be statically decided. So, a further research towards a still more effective implementation of inheritance can be to study the modalities that pertain to large taxonomy.

If the coding need not be bijective, a much shorter coding is available (e.g. sparing set-coding function). It may be useful when ψ-terms act purely as filters and nobody wants to know through which 'holes' the information passed. This is coherent with a logical point of view in which a first order variable stands for any first order term. In LOGIN, a ψ-term, every root symbols in which are not direct antecedents of $\perp$, can be considered as a non-ground term, and can be replaced in a solution by any more precise ψ-term. A non-bijective coding is further supported by the fact that it is convenient (because it is economical) to allow the programmer to give an incomplete signature which is completed internally to a powerset. In this way, many elements of the actual signature have no external representation, and a bijective coding would contradict this convenience. However, it violates the spirit of standard Prolog systems where non-ground terms can be output during the resolution.

The second problem is that the compilation time phase conflicts with the capability, standard in Prolog, to create new symbols. Furthermore, a link editing algorithm for signature must be devised in case of separate compilation.

The use of ordered types improves naturally the expressive power of first order logic programming languages. But it introduces memory management problems that are not solved by classical implementations. It is also the case of other extensions like unification with constraints or fair proof strategy. Unification with constraints meets the same substitution problem as LOGIN, while fair resolution is not compatible with a stack oriented memory management since its proof strategy is not 'left-to-right'. However, these extensions may lead up to a more satisfactory compromise between declarativity and effectiveness than the present compromise of the Prolog dialects. Since MALI is only involved with depth-first search, all these extensions has been successfully implemented with it.

An important feature of MALI is the Attributed Variable. It has two different intuitive understandings corresponding to two classes of applications. The first intuition is that an Attributed Variable is a Variable with a Term fastened to it which is supposed to represent a property of the Variable. This property can be a constraint (Dif or a more general constaint), a type (ψ-term or more classical typing), or a daemon that must be executed to acknowledge some event (Freeze). The second intuition is that an Attributed Variable is a Term —the Attribute— which has the capability of being

Variable. Such a Term can represent a mutable structure. The applications that follow this intuition are the implementation of fair proof strategies or the implementation of sets. Fair proof strategies (excepted the trivial breadth-first strategy) require to be able to select a goal and insert a clause body anywhere in a goal-statement. Thus, mutable structures are needed to avoid to repeatedly reconstruct goal-statements.

It must be noted that MALI and Prolog can be seen as reversible-single-assignment machine and language. They are 'single assignment' because Variables can be substituted only once in the scope of one search. But they are 'reversible assignment' because substitutions can be undone in a LIFO discipline (last done, first undone). The difference between Prolog and MALI is that assignable objects of Prolog (variables) can only be found in a leaf of a data-structure, whereas Attributed Variables of MALI, under the second intuition, are assignable, but non-leaf, objects. The reversible-single-assignment property extended to non-leaf objects plus the connection between substitution and memory management make MALI a powerful tool to implement intricate data structures and execution schemes.

References

[ABLN89] H. Aït-Kaci, R. Boyer, P. Lincoln, and R. Nasr. Efficient implementation of lattice operations. *ACM Transactions on Programming Languages and Systems*, 11(1):115–146, 1989.

[Ait86] H. Aït-Kaci. An algebraic semantics approach to the effective resolution of type equations. *Theoretical Computer Science*, (45):293–351, 1986.

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

[Bak78] H.G. Baker. List-processing in real time on a serial computer. *CACM*, 21(4):280–294, 1978.

[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.

[CLR87] L. Chevallier, S. Le Huitouze, and O. Ridoux. Style de programmation pour une machine de programmation logique munie d'un récupérateur de mémoire. In *Séminaire de Programmation Logique de Trégastel*, CNET, France, 1987.

[Hue76] G. Huet. *Résolution d'équations dans les langages d'ordre 1,2...ω*. Thèse de doctorat d'état, Université de Paris VII, 1976.

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

[RW88] K.A. Ross and C.R.B. Wright. *Discrete Mathematics*. Prentice-Hall, 1988. 2nd edition.

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

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

LISTE DES DERNIERES PUBLICATIONS INTERNES

PI 471 **VERS UNE PROBLEMATIQUE DE L'ALGORITHMIQUE REPARTIE**
JeanMichel HELARY, Michel RAYNAL
12 Pages, Mai 1989.

PI 472 **DERIVATION SYSTEMATIQUE D'UN ALGORITHME DE SEGMENTATION D'IMAGES - UN EXEMPLE D'APPLICATION DU FORMALISME GAMMA**
Christian CREVEUIL, Gersan MOGUEROU
46 Pages, Mai 1989.

PI 473 **MICROCODE OPTIMIZATION FOR THE PCS PROCESSOR**
François BODIN, François CHAROT, Charles WAGNER
26 Pages, Mai 1989.

PI 474 **ALGEBRAICALLY CLOSED THEORIES**
Eric BADOUEL
22 Pages, Mai 1989.

PI 475 **QUELQUES OUTILS GRAPHIQUES POUR LA MODELISATION DU CONTROLE D'EXECUTION EN ROBOTIQUE DE COOPERATION**
Jean-Christophe PAOLETTI, Lionel MARCE
52 Pages, Juin 1989.

PI 476 **SIMULATION REPARTIE DE SYSTEMES A EVENEMENTS DISCRETS: PARTIE 1 : MODELISATION ET SCHEMAS D'EXECUTION**
Philippe INGELS, Michel RAYNAL
26 Pages, Juin 1989.

477 **PROGRAMMING WITH MALI - UNIFICATION OR ORDERED TYPES**
Olivier RIDOUX
18 Pages, Juin 1989.

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