The Compilation of lambdaProlog and its execution with MALI

Pascal Brisset, Olivier Ridoux

The compilation of λProlog and its execution with MALI

Pascal BRISSET
Olivier RIDOUX

N° 1831
Janvier 1993

PROGRAMME 2

Calcul Symbolique,
Programmation
et Génie lociciel

IRISA

INSTITUT DE RECHERCHE EN INFORMATIQUE
ET SYSTEMES ALEATOIRES

Campus Universitaire de Beaulieu
35042 - RENNES CEDEX FRANCE
Tél. : 99 84 71 00 - Télex : UNIRISA 950 473 F
Télécopie : 99 38 38 32

The Compilation of λProlog and its Execution with MALI[1]

Pascal Brisset IRISA — Olivier Ridoux IRISA/INRIA
{brisset, ridoux}@irisa.fr

Abstract — We present a compiled implementation of λProlog that uses the abstract memory MALI for representing the execution state.

λProlog is a logic programming language allowing a more general clause form than Standard Prolog's (namely hereditary Harrop formulas instead of Horn formulas) and using simply typed λ-terms as a term domain instead of first order terms. The augmented clause form causes the program (a set of clauses) and the signature (a set of constants) to be changeable in a very disciplined way. The new term domain has a semi-decidable and infinitary unification theory, and it introduces the need for a β-reduction operation at run-time.

MALI is an abstract memory that is suitable 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 translation along which predicates are transformed into functions operating on continuations for handling failure and success in unifications, and changes in signatures and programs.

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

La compilation de λProlog et son exécution avec MALI

Résumé — Nous présentons une implémentation compilée de λProlog qui utilise la mémoire abstraite MALI pour représenter l'état d'exécution.

λProlog est un langage de programmation logique dont la forme des clauses est plus générale que celle de Standard Prolog (des formules héréditaires de Harrop au lieu de formules de Horn) et dont le domaine de termes est celui des λ-termes simplement typés au lieu des termes de premier ordre. Le nouveau langage de clauses fait que le programme (une collection de clauses) et la signature (une collection de constantes) peuvent changer, mais pas de manière arbitraire. Le nouveau domaine de termes a une théorie de l'unification semi-décidable et infinitaire, et il nécessite d'effectuer des β-réductions pendant le calcul.

MALI est une machine abstraite qui permet de représenter l'état d'un processus de recherche en profondeur. Sa principale caractéristique est une gestion de mémoire efficace.

Nous avons utilisé un schéma original de traduction de λProlog vers C, où les prédicats sont transformés en des fonctions manipulant des continuations pour gérer les échecs et succès de l'unification, et les changements de signature et de programme.

Deux mots-clés de cette implémentation sont "partage" et "superposition" de représentation. Le partage consiste à reconnaître que la représentation d'un terme existe déjà et à la réutiliser. La superposition consiste à reconnaître que deux représentations distinctes sont celles du même terme, et à remplacer l'une par l'autre.

[1] A previous version of this report served as lecture notes for the Logic Programming Winter School and Seminar, Czechoslovakia, January 27–31, 1992 [68].
Comments are welcome.

Introduction

Though one usually carefully distinguishes between *logic programming* and *Prolog*, the research done in logic programming is often about extending Prolog. It is seldom the case that such work leads to a reassessment of the foundation of logic programming. One of the important "meta-features" of λProlog is that it leads to such a reassessment.

The basic idea behind Prolog is that a particular subset of predicate calculus, the logic of Horn formulas, is suitable for programming: every computable relation can be expressed in Horn formulas, and their proof theory yields a convenient execution mechanism.

The theory of Prolog insists on model-theoretic results [45]. The main result is that models of Horn programs are closed under set-intersection. Horn programs have then a unique minimal model, which is the least fixed-point of an immediate consequence function, and can also be enumerated by a search-procedure.

When analysing what makes a proof theory yield a convenient execution mechanism, it appears that richer subsets of predicate logic can also be suitable for programming. The results that are important now are proof-theoretic. The logic of hereditary Harrop formulas is such a subset. In this case, the enrichment is to allow implications and explicit quantifications where Horn formulas only allow conjunctions of atoms.

A pleasant feature of Prolog is that its term language, the language of first-order terms, is rich enough for encoding directly its formula language and manipulating it easily. This makes meta-programming much easier in Prolog than, say, in C. In fact, a closer look at Prolog shows that its quantifications, though implicit, are not properly represented by first-order terms. It appears that the same term language is certainly rich enough for encoding and manipulating hereditary Harrop formulas but it is still done less properly, mainly because of the scoping aspects of quantification. The necessary enrichment is to allow abstractions in terms.

We call Standard Prolog the language of Horn formulas plus first-order terms and λProlog the language of hereditary Harrop formulas plus simply typed λ-terms. We use the word Prolog when we do not insist on a particular language.

New language — new hopes

The extension of Prolog to hereditary Harrop formulas and λ-terms has been proposed by Miller and Nadathur [52].

The advantages of this extension have been advocated by Felty, Miller, Nadathur and others [49, 53, 25, 23, 47, 24]. They range from technical (but very important) issues of symbols manipulation programming (scoping, substitutivity) to software technology concerns (naming, modularity). These advantages make the implementation of other logics in λProlog an easier task than in Standard Prolog.

Example 1
A typical example of λProlog programming is the implementation of a type system. The type system, here the theory of simple types, is a logic characterised by a set of deduction rules.

The system of deduction rules for the theory of simple types is the following:

$$\frac{\Gamma \vdash t_1 : \alpha \rightarrow \beta \qquad \Gamma \vdash t_2 : \alpha}{\Gamma \vdash (t_1\, t_2) : \beta} \quad \rightarrow_E$$

$$\frac{\Gamma, x : \alpha \vdash E : \beta}{\Gamma \vdash \lambda x.E : \alpha \rightarrow \beta} \quad \rightarrow_I \quad \dagger$$

$$\frac{}{\Gamma, x : \tau \vdash x : \tau} \quad \mathit{axiom}$$

† *Variable x does not occur in Γ.*

The right-hand side of $\vdash$ is a type assignment (e.g. term x has type τ), and the left-hand side is a set of type assignments that serves as a typing context. Informally, the rules have the following reading.

$\rightarrow_E$: *application $(t_1\ t_2)$ has type β in typing context Γ if terms t_1 and t_2 have types $\alpha \rightarrow \beta$ and α in Γ.*

$\rightarrow_I$: *abstraction $\lambda x.E$ has type $\alpha \rightarrow \beta$ in typing context Γ if E has type β in a new typing context made of Γ augmented by the assignment of type α to variable x (variable x must not be already assigned a type in Γ because typing contexts must establish a functional relation between variables and types)*

axiom: *every type assignment of a typing context is true in that context.*

The λProlog program that implements this logic is the following[2]*:*

```
has_type  (application T1 T2)  Beta :-
        has_type  T1  (arrow Alpha Beta),
        has_type  T2  Alpha.
has_type  (abstraction E)
          (arrow Alpha Beta) :-
        pi x\
        (   has_type  x  Alpha
        =>  has_type  (E x)  Beta
        ).
```

A sample query is

```
  ?- has_type  (abstraction x\x)  (arrow i i).
```

Relation has_type *is defined by two clauses, which are built with the* :- *connective (read "if"). The first clause encodes rule $\rightarrow_E$, and the second one encodes rule $\rightarrow_I$.*

*The second clause displays almost all λProlog features: explicit universal quantification in goals (*pi x\ *reads "for all x"), intuitionistic implication in goals (*=> *reads "implies"), and λ-terms (*E *is some λ-abstraction and* (E x) *denotes the application of* E *to* x*). In the query,* x\x *reads "lambda x x". The universal quantification encodes the side condition of rule $\rightarrow_I$, the implication encodes the augmentation of the context Γ, and the application encodes the stripping of the abstraction.*

The operational reading of a clause like the first one is the same as in Standard Prolog: a clause

```
p A0 :- q A1, r A2.
```

answers a call to predicate p *by unifying the actual arguments and* A0*, and then calling predicates* q *and* r *(in this order) with arguments* A1 *and* A2*.*

The operational reading of a clause like the second one owes almost nothing to Standard Prolog: a clause

```
p A0 :- pi x\( q (A1 x) => r (A2 x) ).
```

answers a call to predicate p *by unifying the actual arguments and* A0*, and then creating a new constant, say* c*, adding clause* (q (A1 c)) *to the program, and calling predicate* r *with argument* (A2 c) *in the augmented program.*

Rule axiom *has no statically defined encoding. At a given time, it is encoded in extension by the clauses that are added to the program.*

A constant aspect of the advantages of λProlog is that they give a logical notation to phenomena that, in Standard Prolog, ought to be described operationally. E.g. the relation between an abstraction E and one of its instances T is denoted by T = (E x). In Standard Prolog, it has to be expressed by a duplication procedure.

New language — new implementations

All this would be very nice, if it could be implemented in an efficient way. Efficiency can be thought of in two dimensions. First dimension is to make the complexity of the implementation non-deceptive. That is, when a new feature is presented as a logical alternative to an old operational one, it should yield an equal or better complexity. For instance, we propose a function-list notation as a logical alternative for difference-lists [15]. The trick is easy (it is inherited from functional programming), but to make it compete with difference-lists, one must be especially careful in implementing the λ-terms. Another example is the advocated use of intuitionistic implication for implementing local declarations and modularity. It should not add an undue complexity where one is used to access symbols in a constant time. Second dimension is to make everything more efficient via a suitable analysis. This is compilation.

A prototype implementation of λProlog has been done by Miller and Nadathur in Standard Prolog. It was intended for experimental use and is very inefficient. Then, a more robust implementation has been done in Lisp (eLP, by the Ergo Project at Carnegie-Mellon University). The study of a compiled implementation of λProlog has been initiated by Nadathur and Jayaraman [58, 36] and is continuing with Nadathur, Kwon and Wilson [40]. We know only progress reports about it, but no performance report. Finally, Felty and Gunther (Bell Labs) are working on another implementation in ML for extending the capability of ML as a meta-language for automatic theorem proving.

We propose another implementation in which we apply two techniques that we have previously developed for Standard Prolog. We translate λProlog programs into programs of an imperative language (incidentally C), and we use MALI [8, 9, 69] as a term oriented abstract memory. A first version of our compiled implementation of the core of λProlog is now available.

We have worked in the two dimensions of efficiency at the same time but cannot claim to have exhaustively completed even the first one. For instance, our implementation of intuitionistic implication is still deceptive in the above sense. It cannot be used for local declarations without introducing undue complexity. This is not a consequence of some of our technical choices; it is only the current

[2]Type declarations are missing in this program. They are given in example 1.1.1.
The syntax and semantics of λProlog are exposed in the body of this report. To understand this example, it is enough to know that the case convention for identifiers is the same as for Standard Prolog: identifiers beginning with a high-case letter are identifiers of logical variables.

state of our research progress.

A running example: representations of lists

We use as a running example the manipulation of lists in λProlog. It starts with well-known Standard Prolog representations for list, and it is continued by a functional representation as soon as the basics of λProlog are given.

Though this initial part of the example could be written in Standard Prolog, we use λProlog syntax in the program samples. The lexical conventions are the same in Standard Prolog and λProlog. Only construction of terms and goals is different: λProlog uses a "Curryfied" notation. An n-ary term, or goal, constructor is considered as a unitary function that returns a *(n-1)*-ary function. The global lay-out is the same, but it uses much less parenthesis and commas than Standard Prolog.

Prolog lists

When lists are represented by Prolog lists, a *cons* is represented by the binary functor `'.'` (noted `[ | ]`) and the empty list is represented by `nil` (noted `[]`). For instance, $cons(1, cons(2, cons(3, nil)))$ is represented by `1.2.3.nil` (noted `[1,2,3]`).

The Prolog representation of lists is used in the classical list manipulation predicates: "append", "naive reverse", "append 3 lists", and "member":

```
append  []  Y  Y.
append  [E|X]  Y  [E|Z] :-
        append  X  Y  Z.

nrev  []  [] .
nrev  [A|X]  Y :-
        nrev  X  RX,
        append  RX  [A]  Y.

append3  A  B  C  ABC :-
        append  A  BC  ABC,
        append  B  C  BC.

member  E  [E|_] .
member  E  [_|L] :-
        member  E  L.
```

A *mode* is a specification of a particular calling convention. A mode expression is a literal in which terms are replaced by "+" (always ground), "-" (always a logical variable) and "?" (do not know). Among the four predicates above, **append** is the only one which can operate in every mode. Predicate **nrev** enters a loop after issuing a solution for modes `(nrev - +)`. Predicate `(append3 - - - +)` works nice in modes `(append3 + + + -)` and `(append3 - - - +)`, which covers probably the intended usage. Note that the order in which lists `A`, `B` and `C` are composed in **append3** must be carefully chosen for working in the two modes.

Note that predicate **append** can concatenate a list to itself.

```
twice  L  LL :-
        append  L  L  LL.
```

From Prolog lists to difference-lists

In logic programming, the use of incomplete structures is a well-known technique. It is even a part where the logic programming paradigm is at its best. One example of this technique is the difference-list (noted `List-SubList` where "-" is an infix operator). A list is represented by the difference between a Prolog list and one of its sublists (the *tail* of the list) that must be a logical variable[3]. For instance, the empty list is represented by `X-X` and $cons(1, cons(2, cons(3, nil)))$ is represented by `[1,2,3|X]-X`.

Because the tail is a logical variable, two lists can be concatenated with only one unification which is a binding of the tail. Because the left concatenand is not duplicated when its tail is being substituted by the right concatenand, a difference list can be the left concatenand of only one list in its life.

```
dappend  A-ZA  ZA-ZB  A-ZB .

dappend3  A-ZA  ZA-ZB  ZB-ZC  A-ZC .
```

Note that, unlike predicates **append**, predicate **dappend** cannot be used for splitting a list. Nothing in Prolog enforces the constraint that the tail is a sublist of the list. So, unification binds `ZA` to something which is neither a sublist of `A` nor a superlist of `ZB`. Verifying the constraint can only result from a programming discipline. Another difficulty is that testing the empty list requires a unification procedure with an occurrence-check because otherwise Prolog is always willing to unify `[...|Z]-Z` and `X-X`. Similarly, the lack of an occurrence-check makes an attempt to concatenate a difference-list to itself succeed and produce an infinite term.

For all these reasons, the following predicates are bogus.

```
dtwice  L  LL :-
        dappend  L  L  LL.

common_prefix  P  L1  L2 :-
        dappend  P  _  L1,
        dappend  P  _  L2.
```

[3]Logically speaking, the tail could be any list. But, the practical interest of difference-lists relies on the assumption that it is a logical variable.

However, transformation of the representation of lists, followed by some partial evaluation [72], generally leads to efficient programs. Predicate **nrev** can be transformed into the following predicates:

```
drev1  []  Y-Y .
drev1  [A|L]  Z-Y :-
        drev1  L  Z-[A|Y].

drev  L  RL  :-
        drev1  L  RL-[].
```

The explicit representation of the difference-list is often omitted on the ground of the cost of manipulating the '-' term constructor.

```
drev1  []  Y  Y .
drev1  [A|L]  Z  Y :-
        drev1  L  Z  [A|Y].

drev  L  RL  :-
        drev1  L  RL  [].
```

Transforming difference-lists into Prolog lists is trivial (if it is allowed to be destructive like **dappend**), but the way back needs to use predicate **append**.

```
dlist2list  L-[]  L.

list2dlist  L  AL-ZL :-
        append  L  ZL  AL.
```

In λProlog, there is yet another "natural" representation for lists. It merges the logical robustness of standard lists, and the conciseness of difference-lists. We resume the "running example" after presenting the basics of λProlog (see section 1.5), and describe this new representation for lists.

Structure of the report

This report is made of four parts:

1. An account of λProlog syntax and semantics, with practical insight.
2. A description of two basic technical choices: first, to use a memory management machine called MALI (this solves many low-level representation problems at once: mainly, memory management and undoable substitutability), second, to translate source programs into an imperative programming language (incidentally C).
3. The implementation with MALI of an abstract machine for executing Prolog, and a systematic exploration of λProlog features.
4. The features and performances of a λProlog system based on the principles presented in previous sections.

An index allows fast cross-referencing.

Contents

Chapter 1

λProlog

We present in this part the various features of λProlog. First section describes the term domain of λProlog and the associated unification problem. Second section presents the formula domain of λProlog. Third section describes an alternative term domain that is closer to the usual first-order domain. Last two sections contain informal remarks on the pragmatics of λProlog, and the continuation of the "running example".

1.1 Extension of the term domain: simply typed λ-terms

Any extension of the computation domain of Prolog is required to specify what is *unification* for the new terms. The restriction of λ-calculus to Church's simply typed λ-terms [17] allows for a usable definition of the unification of λ-terms (λ-unification). Huet gives an extensive presentation of λ-unification [32]. Paulson gives the main lines of it in the context of theorem proving [62], and Snyder and Gallier revisit it in a deduction rule setting [71]. Nipkow extends the deduction rule setting towards simple types with type variables [60], which is in fact the domain of λProlog. Miller studies the correspondence between logical quantifications (∀ and ∃) and λ-quantification [51].

One can see the extension of Prolog with simply typed λ-terms merely as the instance $\mathcal{CLP}(\lambda_{\rightarrow})$ of the $\mathcal{CLP}$ scheme [35].

Other possible domains in the λProlog flavour are the typed λ-calculi of Barendregt's cube [5, 30]. Elliot and Pfenning have studied unification in some of these calculi [21, 65].

We describe the new domain, some precisions on the notions of variables and substitutions in the new domain, and the unification algorithm.

1.1.1 Simply typed λ-terms

We give a quick introduction to the λ-calculus (see Barendregt or Revesz for more information [6, 67]).

1.1.1.1 Types

Simple types, $\mathcal{T}$, are first-order terms built from a collection of type constants (*type constructors*) and one dedicated binary type constant, $\rightarrow$ ("->" in concrete syntax). Constant $\rightarrow$ is given an infix notation and is supposed to associate to the right.

Types constructed with type constant $\rightarrow$ can be interpreted as types of functions. We call *result type* the rightmost subtype of a type. We call *primitive* a type that is not constructed with type constant $\rightarrow$.

In λProlog, the nullary constant o is reserved as the type of truth values. In every concrete implementation of λProlog, several other type constants are reserved (e.g. `int` and `list`). Every type whose result type is o is called a *predicate type*.

Example 1.1.1
The following type declarations were missing in example 1.

```
kind (lambda_term, simple_type)
       type.
type application
       lambda_term ->
       lambda_term -> lambda_term.
type abstraction
       (lambda_term -> lambda_term) ->
       lambda_term.
type arrow
       simple_type ->
       simple_type -> simple_type.
type has_type
       lambda_term -> simple_type -> o.
```

First declaration introduces type constants `lambda_term` *and* `simple_type`[1].

Second (resp. fourth) declaration introduces `application` *(resp.* `arrow`*) as a (curryfied) binary function on terms of type* `lambda_term` *(resp.* `simple_type`*).*

Third declaration introduces `abstraction` *as a unary function from unary functions on terms of type* `lambda_term`*, to terms of type* `lambda_term`.

[1]The syntax for declaring at once several constants of same kind or type is proper to our implementation of λProlog.

Last declaration introduces `has_type` *as a binary function from terms of type* `lambda_term` *and* `simple_type`*, to terms of type* `o`*. Since type constant* `o` *is conventionally used for truth-values, function* `has_type` *can be interpreted as a binary relation.*

1.1.1.2 Terms

Simply typed λ-terms, Λ, are built from a collection of *constants* (also known as *term constructors*), $\mathcal{C}$, a collection of *λ-variables*, $\mathcal{V}$, a collection of *logical variables* or *unknowns*, $\mathcal{U}$, using the *abstraction* and the *application* rules.

- Sets $\mathcal{C}$, $\mathcal{V}$ and $\mathcal{U}$ are disjoint subsets of Λ.
- If x is a λ-variable and E is in Λ, then $\lambda x.E$ (or `x\`E in concrete syntax) is an abstraction in Λ.

  Nested abstractions $\lambda a_1 \ldots . \lambda a_n.t$ are written $\lambda a_1 \ldots a_n.t$, $\lambda \overline{a_n}.t$, or $\lambda \overline{a}.t$ if the number of individuals does not matter. We call t and $\overline{a}$ the *body* and *binder* of abstraction $\lambda \overline{a}.t$.

  Abstraction is a kind of quantification: the λ-quantification. It has strong connections with universal quantification which are developed in the sequel. As a quantification, abstraction gives rise to the usual notions of *free* and *bound* λ-variables.
- If E and F are in Λ, then $(E\ F)$ is an application in Λ.

  Application is supposed to associate to the left so that nested applications $(\ldots((a_1\ a_2)\ a_3)\ldots a_n)$ are written $(a_1\ a_2\ a_3 \ldots a_n)$, $(\overline{a_n})$, or $(\overline{a})$ if the number of individuals does not matter[2].
- There is a typing function τ from Λ to $\mathcal{T}$ that verifies rules
  $\tau(\lambda x.E) = \tau(x) \rightarrow \tau(E)$
  and
  $\exists\alpha.\ (\tau(E) = \alpha \rightarrow \beta) \wedge (\tau(F) = \alpha)$
  $\Leftrightarrow$
  $\tau((E\ F)) = \beta$

A constant that is given a predicate type is called a *predicate constant*.

Example 1.1.2
A λ-term $\lambda x.E$ with type $\alpha \rightarrow \beta$ can be interpreted as a function with parameter x of type α and result E of type β. For instance, $\lambda x.x$ with type $\alpha \rightarrow \alpha$ is the identity function for terms having type α. It is noted $\mathbf{id}_\alpha$. *Concrete syntax for $\lambda x.x$ is* `x\x`.

Example 1.1.3
For every λ-term f of type $\alpha \rightarrow \alpha$, we note f^0 for $\mathbf{id}_\alpha$, and f^n for $\lambda x.(f\ (f^{n-1}\ x))$. For every integer n, $N_n = \lambda sz.(s^n\ z)$ is a λ-term. It has type $(\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha$. We have

$$\begin{array}{rcl} N_0 & = & \lambda sz.(\mathbf{id}_\alpha\ z) \\ N_1 & = & \lambda sz.(\lambda x.(s(\mathbf{id}_\alpha\ x))z) \\ N_2 & = & \lambda sz.(\lambda x.(s((\lambda x.(s(\mathbf{id}_\alpha\ x)))x))z) \end{array}$$

In concrete syntax:

```
N0 = s\z\(ID z),
N1 = s\z\(x\(s (ID x)) z),
N2 = s\z\(x\(s ((x\(s (ID x))) x)) z)
```

1.1.1.3 Axioms

Three equivalence relations are defined on Λ. The smallest congruence on the structure of terms in Λ that is compatible with the three equivalence relations is called *λ-equivalence* (noted $=_\lambda$).

1. *α-equivalence*, $\lambda x.E =_\alpha \lambda y.E[y/x]$ (variable y is not free in E), defines consistent renaming of λ-variables[3].
2. *β-equivalence*, $(\lambda x.E\ F) =_\beta E[F/x]$ (term F has no free variable bound in E), formalises the application of a function to a term. Application $(\lambda x.E\ F)$ is called a *β-redex*.
3. *η-equivalence*, $\lambda x.(E\ x) =_\eta E$ (variable x is not free in E), formalises extensionality of λ-defined functions[4]. Abstraction $\lambda x.(E\ x)$ where variable x is not free in E is called an *η-redex*.

Example 1.1.4
$(\mathbf{id}_\alpha\ T) =_\beta T$ *for every λ-term T of type α,*

Example 1.1.5

$$\begin{array}{llll} N_0 & =_\lambda \lambda sz.z & & = \lambda s.\mathbf{id}_\alpha \\ N_1 & =_\beta \lambda sz.(s\ z) & =_\eta \lambda s.s & = \mathbf{id}_{\alpha\rightarrow\alpha} \\ N_2 & =_\lambda \lambda sz.(s\ (s\ z)) & & \end{array}$$

The equivalence axioms may be oriented to be considered as *reduction rules*. To apply β-equivalence for suppressing a β-redex is to β-reduce a term. To apply η-equivalence for suppressing a η-redex is to η-reduce a term. The opposite operation is η-expansion.

[2]Note that $\overline{a}$ only denotes a finite sequence of something. It requires a context (λ or ()) to give the interpretation of the sequence. For instance, $\overline{a}$ alone is not an application, but $(\overline{a})$ is. We use $\overline{a_n}$ for a sequence of n somethings.

[3]$[A/B]$ denotes the function that substitutes A for B. $C[A/B]$ denotes a term similar to C, except that every free occurrence of B is replaced by A.

[4]It is trivial that equal functions yield equal results. Extensionality of functions states that functions yielding equal results are equal.

A given term may be applied different reduction rules simply because it may contain several redexes. One may wonder whether the reduction order is significative or not. In fact, it is not. This system of reduction rules (with or without η-reduction, with or without simple types), enjoys the *Church-Rosser property*:

> *For every two β-equivalent terms A and B, there is a term N to which A and B reduce in some number of steps.*

1.1.1.4 Normal forms

A λ-term with no β-redex (resp. η-redex) is called *β-normal* (resp. *η-normal*). Every term of the simply typed λ-calculus has a β-normal form. Because of the Church-Rosser property, it is unique. Moreover, it can be computed using any arbitrary strategy (unlike in the pure λ-calculus). The last property is called the *strong normalisation property*.

Example 1.1.6
In pure λ-calculus, some terms have no normal form: e.g. the combinator $\Omega = (\lambda x.(x\ x))(\lambda x.(x\ x))$ is not in β-normal form, but it only β-reduces to itself.

In pure λ-calculus, some terms with a normal form may have non-terminating reductions: term $((\lambda xy.x)\ \mathbf{id}\ \Omega)$ β-reduces to $\mathbf{id}$ by its first redex, and it β-reduces to itself by the redex in Ω.

Example 1.1.7
In example 1.1.5, the terms of the second column are all β-normal, but $\lambda sz.(s\ z)$ is not η-normal.

We call $\hat{n}$ the β-normal term equivalent to N_n. These λ-terms form the Church's encoding of integers. In fact, many data-types and functions can be encoded in simply typed λ-terms.

In the following, terms are supposed to be in *long head-normal form*: $\lambda\overline{x}.(@\ \overline{t})$ where $@$ is a constant, an unknown or a λ-variable, and $\tau((@\ \overline{t}))$ is primitive. The t's satisfy no special conditions. The term $@$ is called the *head*, $\overline{x}$ is called the *binder*, and $\lambda\overline{x}.@$ is called the *heading*. Long head-normal form is obtained by repeated application of β-reduction to eliminate outermost β-redexes, followed by repeated application of η-expansion to adjust the number of t's to the number of arrows in the type of the head.

Example 1.1.8
In example 1.1.5, the terms of the second column are all in long head-normal form. Term $\lambda s.s$ is not in long head-normal form because the type of s is $\alpha \to \alpha$. Head s expects one more argument.

A term is called *flexible* if its head is an unknown, *rigid* otherwise. A *rigid path* is a path through the syntactic tree of a term that never encounter a flexible term. Other paths are *flexible paths*.

Example 1.1.9
Assuming that U is an unknown and t is constant, $(U\ t)$ is a flexible term and $(t\ (U\ t))$ is not.

In term $(t\ (U\ t))$, there is a rigid path to the leftmost t and a flexible path to the other one.

1.1.2 Generic polymorphism

In λProlog, term constructors are given *type schemes*, which are first-order types with *type variables* for introducing *generic polymorphism* (*à la ML* [55]) in the language. Type *instances* can be obtained by substituting types for type variables in type schemes. Type *renaming* can be obtained by substituting new type variables for type variables in type schemes. Nipkow describes a λ-unification procedure for this kind of types [60].

Polymorphism comes from the use of type variables. Every occurrence of a term constructor has a type which is an instance of its type scheme. Every instance is independent from the others, hence the polymorphism. Every occurrence of the same variable (same identifier, same scope) has the same type.

Example 1.1.10
Follow the necessary[5] *type declarations for predicate* **append***:*

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

First line declares type constructor `list` *and assigns arity 1 to it. In λProlog, only strictly "flat" arities are allowed. So, only the number of arrows matters.*

Next two lines declare constants `[]` *and* `'.'`*. Type variables in the declaration of constant* `'.'` *mean that the types of all its occurrences in the program clauses must be instances of* `A -> (list A) -> (list A)`*.*

Last line declares polymorphic predicate constant **append**.

Example 1.1.11
Follow the type declarations for pairs:

[5] Type constructor `list` and term constructors `[]` and `'.'` are usually predefined. So, these declarations are provided by the system.

```
kind pair_type
        type -> type -> type.
type pair
        A -> B -> (pair_type A B).
```

1.1.3 Substitution and unknowns

In Huet's paper, unification is not embedded in the execution scheme of a logic programming language. So, *substitutions* are defined to operate on free λ-variables of the terms they are applied to. Substitutions are restricted to replace free λ-variables of some types by terms of *the same* types. However, in λProlog there can be no free λ-variable in unification problems hence no substitution in the above sense.

In λProlog[6], substitutions are applied to clauses and goals, and they operate on their free variables only. Variables that are free in unification problems are logical variables. They are in fact implicitly universally quantified at the clause level. λ-variables are all bound in explicit abstractions, whereas logical variables are all bound in outermost implicit universal quantifications.

It appears that logical variables and λ-variables deserve very different implementations. Logical variables pertain to the Prolog technology, whereas λ-variables pertain to the λ-calculus technology. In unification problems, λ-variables behave like universally quantified terms; we say they are *essentially universally quantified.* Symmetrically, logical variables behave like existentially quantified terms; we say they are *essentially existentially quantified*[7].

Now that unknowns and λ-variables are distinguished, the absence of free occurrences of one kind or the other must also be distinguished. We call *closed* a term in which no λ-variable occurs free, and *ground* a term in which no unknown occurs free.

A very important pragmatic property of substitutions on λ-terms is that substitutions may make subterms of a flexible term disappear. This property causes any attempt to make a decision based on the occurrence of some subterm (e.g. occurrence-check) dependent of the substitutions to come. In other words, such decision-making can only be conservative when flexible terms are involved.

[6]In Prolog also, in fact.

[7]In the following, "unknowns", "logical variables" and "essentially existential variables" are strictly synonymous. The only differences are on connotations: "essentially existential variable" recalls the proof theory of quantification, "logical variable" recalls the connection with Standard Prolog variables, and "unknown" recalls the intuition (and allows shorter expressions).

We adopt the convention that identifiers of essentially existential variables begin with a capital letter. This is a natural extension to the Standard Prolog lexical convention. It is not enforced by the syntax: explicitly quantified variables may have any identifier.

Example 1.1.12
Term $(U\ 1\ 2)$ has subterms 1 and 2 and term $(U\ 1\ 2)[\lambda xy.x/U]$ has 1 as only subterm.

1.1.4 λ-unification

A λ-unification problem is characterised by a pair of λ-terms with equal types. If the types are not equal, the problem is *ill-typed.* To unify two simply typed λ-terms t_1 and t_2 is to find a substitution σ such that $\sigma t_1 =_{\alpha\beta\eta} \sigma t_2$.

A variant λ-unification problem can be defined without η-equivalence. The corresponding unification procedure is only more complicated than with η-equivalence. Moreover, η-equivalence is natural in the context of λProlog: it makes λ-abstraction and ∀-quantification dual concepts on terms and formulas. This is generalised in the notion of essentially universal variable, and it is studied more deeply in sections 1.4 and 3.2.6.

With or without η-equivalence, the problem is semi-decidable and infinitary. If there are solutions, they can be found, but if not, the search may diverge. There may be several most general solutions, sometime infinitely many, however they can always be enumerated.

Example 1.1.13
Terms $(F\ 1)$ and 1, where $F \in \mathcal{U}$, are unified by both $\sigma_1 = [\mathbf{id}_{\mathbf{int}}/F]$ and $\sigma_2 = [\lambda x.1/F]$. But neither σ_1 nor σ_2 is more general than the other.

Example 1.1.14
Terms $\lambda v.(F\ v)$ and $\lambda v.v$, where $F \in \mathcal{U}$ and $\tau(F) = \mathbf{int} \rightarrow \mathbf{int}$, are unified by $\sigma_1 = [\mathbf{id}_{\mathbf{int}}/F]$. This example shows the role of λ-variables. λ-variable v takes the place of 1 in example 1.1.13, but it (its name) cannot be captured by a substitution. Hence, there is no solution such as $\sigma_2 = [\lambda x.v/F]$.

Example 1.1.15
Terms $\lambda z.(N\ \mathbf{id}_\alpha\ z)$ and $\mathbf{id}_\alpha$, where $N \in \mathcal{U}$, are unified by every $[\hat{\imath}/N]$, where $\hat{\imath}$ is the Church's encoding of integer i.

1.1.4.1 Search procedure

Huet's algorithm is a search procedure in a tree in which every node is a unification problem and every arc is labelled by an elementary substitution. Terminal nodes are *success nodes* (an empty unification problem) or *failure nodes* (a trivially unsolvable unification problem).

The invariant of the tree is that the compositions of the elementary substitution labelling an arc from node A to node B with the solutions to B are solutions to A. The composition of all the elementary substitutions on the path from the root to a success node is a solution to the root unification problem.

```
SIMPL: $(\Lambda \times \Lambda) \rightarrow (2^{(\Lambda\times\Lambda)} \cup$ Failure$)$
SIMPL$(< t^1, t^2 >) =$
        assume $t^1 = \lambda\overline{u}.(@_1\ \overline{e^1_{p_1}})$ and $t^2 = \lambda\overline{v}.(@_2\ \overline{e^2_{p_2}})$
        in if $@_1 \in \mathcal{U}$
           then $\{< t^1, t^2 >\}$
           else if $@_2 \in \mathcal{U}$
                then $\{< t^2, t^1 >\}$
                else if $@_1 \neq ((\lambda\overline{v}.@_2)\ \overline{u})$          comparison modulo $\alpha$-conversion
                     then Failure
                     else $\cup_{i\in[1\ p_1]}$ SIMPL$(< \lambda\overline{u}.e^1_i, \lambda\overline{v}.e^2_i >)$
```

Figure 1.1: Procedure SIMPL

```
MATCH: $(\Lambda \times \Lambda) \rightarrow (\mathcal{U} \rightarrow \Lambda)$
MATCH$(< \lambda\overline{x}.(F\ \overline{s_p}), \lambda\overline{y}.(@\ \overline{t_q}) >)$
        choose
        when $@ \in \mathcal{C}$
             then $[\lambda\overline{u_p}.(@\ \overline{E_q})/F]$                          imitation rule
        when $@ \in \mathcal{C} \cup \mathcal{V}$ and $\tau(s_i) = \tau_1 \rightarrow \ldots \tau_m \rightarrow \tau((F\ \overline{s_p}))$, $i \in [1\ p]$
             then $[\lambda\overline{u_p}.(u_i\ \overline{E_m})/F]$                        projection rule
```

Every E_k in $\overline{E_q}$ or $\overline{E_m}$ stands for $(H_k\ \overline{u_p})$, where H_k is a new unknown with the appropriate type.

Figure 1.2: Procedure MATCH

$$\begin{tikzcd}
 & & {< (X\ \lambda xy.x\ \ t\ \ e), t >} \arrow[dl, "{[\lambda xyz.(x\ (H_1\ x\ y\ z)\ (H_2\ x\ y\ z))/X]}"'] \arrow[d, "{[\lambda xyz.y/X]}"] \arrow[dr, "{[\lambda xyz.z/X]}"] \arrow[drr, "{[\lambda xyz.t/X]}"] & & \\
 & {< (H_1\ \lambda xy.x\ \ t\ \ e), t >} \arrow[dl, "{[\lambda xyz.(x\ (H'_1\ x\ y\ z)\ (H'_2\ x\ y\ z))/H_1]}"'] \arrow[d, "{[\lambda xyz.y/H_1]}"] \arrow[dr, "{[\lambda xyz.z/H_1]}"] \arrow[drr, "{[\lambda xyz.t/H_1]}"] & \mathit{success} & \mathit{fail} & \mathit{success} \\
\ldots & \mathit{success} & \mathit{fail} & \mathit{success} &
\end{tikzcd}$$

Figure 1.3: Example of a unification tree

Unification problems in the search-tree are supposed to be in *simplified* form (a set of pairs of long head-normal form terms which are not both rigid). Simplified form is obtained by a recursive descent in the structure of the two terms. Trivial failure is detected during simplification when heads of both terms are rigid and different from each other. All this is done by a procedure called SIMPL that takes a well-typed problem and returns a set of simplified problems or fails. It works like the first-order unification procedure on *rigid-rigid* pairs, but it has to remember the abstraction context.

The pseudo-code in figure 1.1 shows the similarity with a first-order unification procedure. The differences can be found in the handling of abstractions in the recursive call, in the comparison of heads $@_1$ and $@_2$, and in the handling of flexible-rigid pairs. The more complex comparison ($@_1 \neq ((\lambda\overline{v_n}.@_2)\ \overline{u_n})$) copes with constant heads and with λ-variable heads. In the last case, it is their positions in their bindings that count, not their names. With a first-order unification procedure, the occurrence of a flexible-rigid pair triggers a substitution. In the same situation, procedure SIMPL simply returns the problem. Remember that procedure SIMPL does not solve a unification problem; it only simplifies it.

The expansion of a non-terminal node uses a procedure called MATCH (see figure 1.2) which takes one flexible-rigid pair, $< \lambda\overline{x}.(F\ \overline{s_p}), t >$, and returns substitutions for unknown F according to two rules, *imitation* and *projection*. The preconditions of the two rules are not exclusive and the projection rule may have several instances, hence the non-determinism. At most *p+1* substitutions can be produced by the two rules. For every substitution, a child node (in Huet's search-tree) is created by applying the substitution to the parent node and simplifying the result.

Imitation and projection aim at making the heads of the two terms equal it two different ways. The imitation way is to carry the rigid head into the flexible head. The projection way is to suppose that the rigid head occurs in one argument of the flexible term and to search it (projection). In both cases, all the details (the $\lambda\overline{u_p}$. and the E_i's) are about well-typing of the whole thing.

Example 1.1.16
Let e_1 and e_2 be $(X\ \lambda xy.x\ \ t\ \ e)$ and t, with $X \in \mathcal{U}$, $x, y \in \mathcal{V}$, and $t, e \in \mathcal{C}$, and their types be $\tau(x) = \tau(y) = \tau(t) = \tau(e) = \gamma$ and $\tau(X) = (\gamma \rightarrow \gamma \rightarrow \gamma) \rightarrow \gamma \rightarrow \gamma \rightarrow \gamma$, where $\gamma \in \mathcal{T}$ is a type constant.

The matching tree for computing the unifiers of e_1 and e_2 is given in figure 1.3. All substitutions are produced by projections except for two imitations in the rightmost edges. Note that the development of the leftmost branch does not terminate. This shows that the search procedure is subject to strategical choices.

In our implementation of λProlog, the search procedure traverses the tree in a depth-first way. As for Standard Prolog, efficiency is at the cost of the loss of completeness.

Example 1.1.17
The unification problem in example 1.1.13 is already in simplified form. Projection rule produces substitution $\sigma_1 = [\mathbf{id_{int}}/F]$ and imitation rule produces substitution $\sigma_2 = [\lambda x.1/F]$. In both cases, the unification problem reduces to $< 1, 1 >$ which simplifies to an empty set, the success node.

Example 1.1.18
The unification problem in example 1.1.14 is already in simplified form. Projection is as for example 1.1.13 but imitation is no more possible because condition $@ \in \mathcal{C}$ is not satisfied (here $@ = v$ and $v \in \mathcal{V}$).

Example 1.1.19
The unification problem in example 1.1.15 is already in simplified form. Again, imitation is not possible because the rigid head is a λ-variable. There are two possible projections.

1. *Projection on the second argument yields substitution $\sigma_{zero} = [\lambda sz.z/N]$. It substitutes $\hat{0}$ to N. The unification problem reduces to $< \mathbf{id}_\alpha, \mathbf{id}_\alpha >$ which simplifies to an empty set.*
2. *Projection on the first argument yields substitution $\sigma_{succ} = [\lambda sz.(s\ (N_1\ s\ z))/N]$. It reduces to $< \lambda f.(N_1\ \mathbf{id}_\alpha\ f), \mathbf{id}_\alpha >$. This is the original unification problem apart from the name of the logical variable. It can either be solved by a substitution such as σ_{zero}, or be reduced to a new, but similar, problem by a substitution such has σ_{succ}.*

Hence, the Church's encoding of every integer is eventually produced. Note that a bad strategy (projecting first on the first argument) makes the procedure try to search the integers starting from the end! Note that a bad strategy for a problem may be good for another.

This search procedure calls for some remarks.

- A pair $< X, t >$ (X does not occur in t) has a most general unifier: $[t/X]$. A first-order unification procedure produces the solution in one step. The higher-order unification procedure constructs the solution piecemeal. There are as much steps as there are term constructors in t. So, instead of letting procedure MATCH solve these problems expensively, SIMPL calls a procedure, named TRIV, that handles cheaply as many as possible similar cases.

Note that TRIV does much more than finding cheaply a solution that SIMPL and MATCH could find expensively. If TRIV is equipped with an occurrence-check, then it detects failure that SIMPL plus MATCH cannot detect. For instance, MATCH with SIMPL loops on problem $< V, (f\ V) >$, whereas TRIV may detect an occurrence-check failure. Remember however, that occurrence-checking cannot be complete because of flexible applications.

- A flexible-flexible pair is not solved but delayed as a *constraint*. The constraint is tested for satisfiability as soon as the pair becomes more rigid.

  So, what is really computed by the procedure is a *pre-unifier*. The flexible-flexible pairs that remain unsolved in a success leaf are always solvable[8]. It would be a bad idea to try to solve them because they have too many minimal solutions, which are not completely determined by the terms of the problem.

Example 1.1.20
The unification problem in example 1.1.14 can be solved by procedure TRIV. Term $\lambda v.(F\ v)$ is η-equivalent to unknown F. So, the problem is trivial, and the single solution, found by procedure MATCH in example 1.1.18, is computed directly by procedure TRIV.

Types are essential because

1. types make the unification problem well defined,
2. in a non-typed λ-calculus, some terms have no normal form,
3. in a non-typed λ-calculus, some terms with a normal form may have non-terminating reductions, and
4. types are used in the projection operation of MATCH.

1.2 Extension of the clause form: hereditary Harrop formulas

To avoid the word "logic" in "logic programming" being only a catchword, one needs to define precisely what is the relation between logic programming and logic[9]. Miller proposes to define a logic programming language as a fragment of a predicate logic that enjoys a *uniform sequent proof* property [54].

1.2.1 What is in a logic programming language?

A logic programming language is defined by its legal programs (clauses) and queries (goals), $\mathcal{D}$ and $\mathcal{G}$. This defines by induction the legal sequents ($P \vdash Q$, where $P \in \mathcal{D}$ and $Q \in \mathcal{G}$) and the legal deduction rules and axioms.

A sequent proof of a theorem in a given fragment is a tree whose leaves are legal axioms, and whose nodes are instances of legal deduction rules. Furthermore, the conclusion of a child node must be a premise of its parent node. The conclusion of the root node is the theorem.

A sequent proof is uniform if every consequent in it is a singleton, and every non-atomic consequent in it is in the conclusion of an instance of a right-introduction rule. Finally, a fragment of a predicate logic enjoys the uniform sequent proof property if all its theorems have uniform proofs.

The purpose of this definition is to restrict proofs to those that

> *give a procedural reading to the consequents.*

In this way, connectives can be considered as control constructs, and deduction rules can be considered as computation rules. An immediate output of the uniform proof property is that the logic of logic programming is intuitionistic.

Defining the formula domain after some proof-theoretic property such as uniformness can be used in other logics than predicate calculus. For instance, Hodas and Miller define another logic programming language as a fragment of linear logic [26, 29].

1.2.2 Standard Prolog

A well known logic programming language is Standard Prolog (see figure 1.4). The languages of its defining fragment are *definite clauses* ($a_0 \subset a_1 \wedge \ldots \wedge a_n$) for $\mathcal{D}$ and non-empty conjunctions of atoms ($a_1 \wedge \ldots \wedge a_n$) for $\mathcal{G}$. The union of both constitutes *Horn clauses*. The only right-introduction rule is $\wedge_R$.

We call *head* the a_0 part of a clause, and *body* the $a_1 \wedge \ldots \wedge a_n$ part. We call *predicate* a maximal set of clauses whose heads are built with the same symbol.

[8] It is so only if every type is inhabited. When it is not the case, the solvability of flexible-flexible pairs is undecidable [51].

[9] In this section, we assume the elementary knowledge of sequent calculus vocabulary.

Logical syntax:

$$\begin{array}{lcl} \mathcal{D} & ::= & A \mid A \subset \mathcal{G} \mid \forall x.\mathcal{D} \mid \mathcal{D} \wedge \mathcal{D} \\ \mathcal{G} & ::= & A \mid \mathcal{G} \wedge \mathcal{G} \\ A & ::= & \text{Atomic formulas} \end{array}$$

Approximate concrete syntax (universal quantifications are implicit):

$$\begin{array}{lcl} \mathcal{D} & ::= & A \mid A \texttt{ :- } \mathcal{G} \texttt{ .} \mid \mathcal{D}\ \mathcal{D} \\ \mathcal{G} & ::= & A \mid \mathcal{G} \texttt{ , } \mathcal{G} \end{array}$$

Deduction rules:

$$\frac{P_1 \vdash A}{P_1 \wedge P_2 \vdash A} \qquad \frac{P_2 \vdash A}{P_1 \wedge P_2 \vdash A} \qquad \wedge_L$$

$$\frac{P \vdash G_1 \qquad P \vdash G_2}{P \vdash G_1 \wedge G_2} \qquad \wedge_R$$

$$\frac{P \wedge D[t/x] \vdash A}{\forall x.D \in P, P \vdash A} \qquad \forall_L \quad t \text{ is an arbitrary term.}$$

$$\frac{P \vdash G}{A \subset G \in P, P \vdash A} \qquad \supset_L$$

$$\frac{}{A \in P, P \vdash A} \qquad \text{axiom}$$

Figure 1.4: The Horn formulas fragment (Standard Prolog)

Logical syntax:

$$\begin{array}{lcl} \mathcal{D} & ::= & A \mid A \subset \mathcal{G} \mid \forall x.\mathcal{D} \mid \mathcal{D} \wedge \mathcal{D} \\ \mathcal{G} & ::= & A \mid \mathcal{G} \wedge \mathcal{G} \mid \mathcal{G} \vee \mathcal{G} \mid \mathcal{D} \supset \mathcal{G} \mid \forall x.\mathcal{G} \mid \exists x.\mathcal{G} \\ A & ::= & \text{Atomic formulas} \end{array}$$

Approximate concrete syntax (universal quantifications in $\mathcal{D}$ are implicit):

$$\begin{array}{lcl} \mathcal{D} & ::= & A \mid A \texttt{ :- } \mathcal{G} \texttt{ .} \mid \mathcal{D}\ \mathcal{D} \\ \mathcal{G} & ::= & A \mid \mathcal{G} \texttt{ , } \mathcal{G} \mid \mathcal{G} \texttt{ ; } \mathcal{G} \mid \mathcal{D} \texttt{ => } \mathcal{G} \mid \texttt{pi x\textbackslash } \mathcal{G} \mid \texttt{sigma x\textbackslash } \mathcal{G} \end{array}$$

New deduction rules:

$$\frac{P \vdash G_1}{P \vdash G_1 \vee G_2} \qquad \frac{P \vdash G_2}{P \vdash G_1 \vee G_2} \qquad \vee_R$$

$$\frac{P \wedge D \vdash G}{P \vdash D \supset G} \qquad \supset_R$$

$$\frac{P \vdash G[c/x]}{P \vdash \forall x.G} \qquad \forall_R \quad c \text{ is a symbol that appears free neither in } P \text{ nor in } G.$$

$$\frac{P \vdash G[t/x]}{P \vdash \exists x.G} \qquad \exists_R \quad t \text{ is an arbitrary term.}$$

Figure 1.5: The hereditary Harrop formulas fragment (λProlog)

$$\dfrac{\dfrac{\dfrac{\dfrac{}{P \land (\mathbf{app}\ \texttt{[]}\ \texttt{[2]}\ \texttt{[2]}) \vdash (\mathbf{app}\ \texttt{[]}\ \texttt{[2]}\ \texttt{[2]})}\ \text{axiom}}{P \vdash (\mathbf{app}\ \texttt{[]}\ \texttt{[2]}\ \texttt{[2]})}\ \forall_L,\ \text{clause } C_1}{P \land (Q \subset (\mathbf{app}\ \texttt{[]}\ \texttt{[2]}\ \texttt{[2]})) \vdash Q}\ \supset_L}{P \vdash Q}\ \forall_L^4,\ \text{clause } C_2$$

Figure 1.6: A uniform sequent proof in Standard Prolog

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{P \land (\mathbf{type}\ c\ \mathbf{i}) \vdash (\mathbf{type}\ ((\lambda x.x)\ c)\ \mathbf{i})}\ \text{axiom}}{P \vdash ((\mathbf{type}\ c\ \mathbf{i}) \supset (\mathbf{type}\ ((\lambda x.x)\ c)\ \mathbf{i}))}\ \supset_R}{P \vdash \forall x.((\mathbf{type}\ x\ \mathbf{i}) \supset (\mathbf{type}\ ((\lambda x.x)\ x)\ \mathbf{i}))}\ \forall_R}{P \land (Q \subset (\forall x.((\mathbf{type}\ x\ \mathbf{i}) \supset (\mathbf{type}\ ((\lambda x.x)\ x)\ \mathbf{i})))) \vdash Q}\ \supset_L}{P \vdash Q}\ \forall_L^3,\ \text{clause } C_2$$

Figure 1.7: A uniform sequent proof in λProlog

As we have noticed above, any logic programming language is a fragment of an intuitionistic predicate calculus. But in the case of Horn clauses, uniform proofs are also complete for the classical calculus.

Example 1.2.1
Let P be
$C_1 : \forall x.(\mathbf{app}\ \texttt{[]}\ x\ x)$
$C_2 : \forall exyz.((\mathbf{app}\ \texttt{[}e\texttt{|}x\texttt{]}\ y\ \texttt{[}e\texttt{|}z\texttt{]}) \subset (\mathbf{app}\ x\ y\ z))$,
let Q be $(\mathbf{app}\ \texttt{[1]}\ \texttt{[2]}\ \texttt{[1,2]})$,
figure 1.6 shows a uniform sequent proof of $P \vdash Q$.

Rule labelled $\forall_L^4$ is a shortcut for four applications of rule $\forall_L$ in which universally quantified variables e, x, y, and z are replaced by 1, [], [2], *and* [2].

The proof is made in the strict setting of the system of deduction rules for Horn formulas. An application of rule $\forall_L$ seems to need some "magic" to select a term compatible with the remaining of the proof, when the rule only says to select an arbitrary term. It is a nice result of theorem proving theory that selection can be delayed and that unification can be used instead of equality in the axiom rule. So, no magic is required.

Axioms and rules $\forall_L$ and $\supset_L$ are never implemented as such; they are wrapped in the *resolution rule* [70]. The universal quantifications at the clause level (which are implicit in the concrete syntax) introduce logical variables, also called unknowns. Rule $\forall_L$ is implemented by the "renaming of unknowns".

As a matter of fact, rules like $\forall_L$, in which an arbitrary term must be chosen, are often implemented as the introduction of a new unknown. The choice of the term is done lazily, guided by unification. Axioms and rule $\supset_L$ enforce that the A's in the antecedent and the consequent of the conclusion are the same by unifying them. A unifying substitution must be applied to the G in the premise. To prefer the most general unifier can be seen as being as lazy as possible.

The implementation of rule $\land_L$ is responsible for the completeness of the proof search. In almost every implementation of Standard Prolog, an unbounded depth-first search with backtracking is chosen. It loses completeness, but it is easy to implement.

1.2.3 λProlog

The clause language $\mathcal{D}$ of λProlog (see figure 1.5) is the language of *hereditary Harrop formulas*. The clause language and the goal language $\mathcal{G}$ are defined by mutual recursion. λProlog has all the deduction rules of Standard Prolog plus deduction rules for the new connectives. Again, rule $\exists_R$ in which a term is to be chosen is implemented by introducing a new unknown.

Notions of head and body of clauses, and of predicate extend naturally to hereditary Harrop formulas.

Example 1.2.2
Let P be
$C_1 : \forall t_1 t_2 \alpha\beta.(\mathbf{type}\ (\mathbf{app}\ t_1\ t_2)\ \beta)$
$\qquad \subset ((\mathbf{type}\ t_1\ \alpha \rightarrow \beta) \land (\mathbf{type}\ t_2\ \alpha)$
$C_2 : \forall e\alpha\beta.(\mathbf{type}\ (\mathbf{abs}\ e)\ \alpha \rightarrow \beta)$
$\qquad \subset (\forall x.((\mathbf{type}\ x\ \alpha) \supset (\mathbf{type}\ (e\ x)\ \beta)))$,
let Q be $(\mathbf{type}\ (\mathbf{abs}\ \lambda x.x)\ \mathbf{i} \rightarrow \mathbf{i})$,
figure 1.7 shows a uniform sequent proof of $P \vdash Q$.

Rule labelled $\forall_L^3$ is a shortcut for three applications of rule $\forall_L$ in which universally quantified *variables e, α, and β are replaced by $\lambda x.x$,* **i**, *and* **i**.

Note that rule axiom is required to prove $c =_\lambda ((\lambda x.x)\ c)$.

Rules $\forall_R$ and $\supset_R$ are the most interesting because they cause a departure from the implementation of Standard Prolog.

Rule $\supset_R$ augments the antecedent (i.e. the program) of the child sequent. Its procedural semantics is

> *To prove $D \supset G$ with program P, prove G with program $P \wedge D$.*

Rule $\forall_R$ augments the set of constants (i.e. the signature) available for constructing clauses and goals. Note that the lazy way in which rules $\forall_L$ and $\exists_R$ are usually implemented causes a problem with rule $\forall_R$. Since all terms are not ground when rule $\forall_R$ is used, to ensure that the new constant c does not appear in either G or P, one must constrain already existing unknowns to never be substituted by terms containing the new constant. I.e. in context $\ldots\forall x\ldots\exists U\ldots\forall y\ldots$, universal variable x can be captured by unknown U, but y cannot. λ-variables are essentially universal, they are always bound in the rightmost part of the context. So, they cannot be captured by unknown.

"Already existing" should be understood in the goal-directed proof time. I.e. an unknown already exists in a node if it appears in a node closer to the root.

The term domain of λProlog is made of simply typed λ-terms. λ-unification is to be used every time equality of atomic formulas must be enforced (i.e. rule $\supset_L$ and axiom). An atomic formula is obtained by applying a constant with result type o to enough *closed* λ-terms so that the type of the formula is o.

Note that only closed term can be used for building an atomic formula. We call *object terms* those that are allowed to serve as arguments to atomic formulas or as binding values to unknowns. They are the terms on which predication can applies. We show in section 1.4.1 that they are all closed terms.

1.3 A mild extension of the term domain: L_λ

Miller presents a fragment of the simply typed λ-calculus that allows for a simple (decidable and unitary) unification [46].

The fragment L_λ can be considered as a substitute for λProlog, or as defining the circumstances under which unification in λProlog can be implemented more efficiently. Among other things, L_λ is the kind of pattern that procedure TRIV may recognise for improving unification. We also believe that it gives a good insight on the pragmatics of flexible terms.

We give in the following an informal proof that unification in L_λ is unitary and decidable, and some insight on the use of L_λ.

1.3.1 The domain of L_λ

The fragment, which is called[10] L_λ, is defined such that unification modulo rules α, β and η is equivalent to unification modulo rules α, β_0 and η, where rule β_0 is the following: $((\lambda x.E)\ x) =_{\beta_0} E$. So, a procedure solving a unification problem in L_λ (if one exists) can be used instead of Huet's algorithm every time a problem is in L_λ.

An important property of β_0-reduction is that it amounts to renaming λ-variables.

Example 1.3.1
Term $\lambda y.((\lambda x.E)\ y)$ β_0-reduces to $\lambda y.E[y/x]$ via an α-conversion.

As defined by Miller, L_λ has the same formula language as λProlog. This means that it takes into account, among other things, the universal quantifications in goals. We think that it is worth presenting L_λ in a simpler language for pedagogical purpose. Furthermore, one must always remember that universal quantifications and λ-quantifications are essentially similar. We show in section 3.2.6 how to encode one into the other. So, we first present L_λ as a fragment of the language $\mathcal{CLP}(\lambda_{\rightarrow})$, and then we complete the definition in section 1.3.3.

The fragment of the term language can be characterised by a syntactic restriction on flexible applications:

> *In every flexible application of a β-normal L_λ term, all the arguments must be distinct λ-variables.*

The term being in β-normal form, there are no more β-redexes. However, flexible applications are potential redexes that can be "activated" during unification when the flexible heads get substituted. If the syntactic restriction is satisfied, then the new β-redexes are β_0-redexes.

Example 1.3.2
Terms $\lambda x.(+\ x\ x)$ and $\lambda xy.(U\ x\ y)$ are in L_λ (the first term has no flexible subterm, the second one has one flexible subterm and it satisfies the restriction).

Example 1.3.3
Terms $\lambda xy.(U\ \ 1\ \ x\ \ y)$, $\lambda xy.(U\ \ x\ \ y\ \ x)$, and

[10]We use the name L_λ for both the term domain and the logic programming language built over it.

$\lambda xy.(U\ x\ V\ y)$ are not in L_λ. The first term is flexible and contains subterm 1, which is not a λ-variable, the second one is flexible and contains several occurrences of the same λ-variable, and the third term is flexible with an unknown as argument.

Miller shows that this restricted language is powerful enough for coding λProlog in L_λ by a local translation [50].

1.3.2 The intuition of L_λ

The intuition of a flexible L_λ term is the following: $(U\ \overline{x})$ stands for an unknown term in which the x's are the only λ-variables that are allowed to occur. And since there is only one occurrence of every λ-variable x, there is no indetermination on *which* occurrence of a λ-variable is chosen for unifying two terms.

This intuition can be extended to rigid terms saying that they are (partially) unknown terms in which some λ-variables *must* occur and some others *can* occur. λ-variables that must occur are those that have at least one occurrence at the end of a rigid path. λ-variables that can occur are those that always occur at the end of a flexible path[11]. When unifying two L_λ terms, one must make sure that the λ-variables that must occur are the same in the two terms, and one has to build a common instance that accepts the λ-variables that are allowed in both terms.

Example 1.3.4
In term $\lambda uv.(f\ u\ (U\ u\ v)\ (V\ v))$, variable v may disappear because it only occurs at the end of flexible paths, and variable u definitely occur because it occurs at the end of a rigid path.

Substitution $[\lambda ab.(W\ a)/U, \lambda a.X/V]$ suppresses every occurrence of variable v, but no substitution can suppress the rigid occurrence of variable u.

To force a flexible L_λ term to allow less λ-variables is done by substituting to its head (U in the above example) an L_λ term of the form $\lambda\overline{y_p}.(V\ y_{i_1} \ldots y_{i_q})$ in which V is a new unknown and $q < p$. Note that it is impossible to force a flexible L_λ term to accepts more λ-variables. In a rigid term, λ-variables that are arguments in a flexible subterm can be eliminated in the same way. A λ-variable that occurs as an argument of a rigid subterm cannot be eliminated because of the form of the flexible paths in L_λ.

This intuition holds also for the whole term language of λProlog, but it does not yield an algorithm so directly because of the lack of restriction.

[11]Note that in L_λ, a flexible path is always made of a, possibly empty, rigid path followed by one flexible application.

1.3.3 About universal variables and L_λ

The definition of the L_λ fragment of λProlog must be completed for dealing with universal variables. Universal variables that are bound in the scope of an unknown can be arguments of this unknown, whereas universal variables that are bound out of its scope cannot.

So, given an unknown, the universal variables of its scope behave as λ-variables and the others behave as constant terms. This is coherent with the fact that the former cannot be captured by the unknown whereas the latter can.

So, the syntactic restriction becomes:

> *In every flexible application of a reduced L_λ term, all the arguments must be distinct universal variables or λ-variables, and the universal variables must be quantified in the scope of the flexible head.*

Example 1.3.5
Formula $\forall x.\exists Y.(p\ (Y\ x))$ is not a L_λ formula because existential variable Y is applied to a universal variable, x, that is not quantified in its scope. Using the intuition given in section 1.3.2, what is wrong with this term is that variable x has two grounds for occurring in term $(Y\ x)$:

- *variable x is an argument to variable Y,*
- *and, variable Y is allowed to capture variable x because Y is in the scope of x.*

Then, it is redundant, and a cause of indetermination, to apply Y to x.

Formula $\exists Y.\forall x.(p\ (Y\ x))$ is in L_λ.

1.3.4 Unification in L_λ is unitary and decidable

One can show that λ-unification in L_λ is unitary by analysing the behaviour of Huet's procedure when it is applied to terms in L_λ.

Procedure SIMPL behaves as in section 1.1.4.1 until it produces flexible-rigid or flexible-flexible pairs.

Before trying to solve a flexible-rigid pair, one must do an occurrence-check: the flexible head must not occur in the rigid term. Unlike in the general case (see discussion in section 1.1.3), occurrence-check in L_λ can always be decided because an unknown can never hide another unknown (unlike, for instance, $< X, (f\ (U\ X)) >$ which is not in L_λ). Occurrence-checking is not necessary for flexible-flexible pairs, because, by construction, there can be no unknown in their arguments.

1.3.4.1 Unification in L_λ is unitary

The flexible-rigid case

In the flexible-rigid case, it is easy to show that procedure MATCH can choose deterministically between imitation and a single projection.

Let $< \lambda\overline{x}.(F\ \overline{s_p}), \lambda\overline{x}.(@\ \overline{t_q}) >$, with $F \in \mathcal{U}$ and $@ \in \mathcal{V} \cup \mathcal{C}$, be a flexible-rigid pair. From the L_λ restriction, it comes that the s_i's ($i \in [1,p]$) are distinct variables, either taken in $\{\overline{x}\}$, or universally bound in the scope of F.

1. If @ is a constant or a universal variable bound out of the scope of F, the imitation rule produces $[\lambda\overline{u_p}.(@\ \overline{E_q})/F]$.

   Note that the projection rule cannot solve the problem because it can only produce a term with a head s_i. Every s_i is different from @ because they must be λ-variables or universal variables bound in the scope of F (definition of L_λ) whereas @ is a constant or a universal variable bound out of the scope of F (precondition of imitation). So, in this case, the unification problem has a single solution.

2. If @ is neither a constant nor a universal variable bound out of the scope of F, the imitation rule cannot apply. The projection rule aims at selecting arguments of the flexible term that can match the rigid head.

   In L_λ, projection can produce a substitution (possibly) leading to a success-node only when the rigid head is one of the s_i's, and in this case only one substitution solves the problem because there cannot be several occurrences of any of the s_i's. If the rigid head is not one of the s_i's, unification fails. Otherwise, for the unique i such that $@ = s_i$ the projection rule produces $[\lambda\overline{u_p}.(u_i\ \overline{E_q})/F]$. In this case also the unification problem has a single solution.

As in procedure MATCH (see figure 1.2), every E_k in $\overline{E_q}$ stands for $(H_k\ \overline{u_p})$, where H_k is a new unknown with the appropriate type. After the substitutions are applied, the problem is still in L_λ.

The flexible-flexible case

In the flexible-flexible case, Huet's procedure does not decide because the problem admits too many solutions. In L_λ, the problem has only one solution and can be decided easily.

Let $< \lambda\overline{x}.(F\ \overline{s_p}), \lambda\overline{x}.(G\ \overline{t_q}) >$, with F and G in $\mathcal{U}$, be a flexible-flexible pair. It follows from the L_λ restriction that the t_i's ($i \in [1,q]$) are also distinct variables, either taken from $\{\overline{x}\}$, or universally bound in the scope of G.

1. If $F = G$ (then $p = q$ because of the well-typing condition), only variables that are in the same position in the bodies of both terms can be exploited in further bindings. Let $\overline{i_k}$ be the i's such that $s_i = t_i$, the solution substitution is $\sigma = [\lambda\overline{u_p}.(H\ u_{i_1} \ldots u_{i_k})/F]$.

   Note that any permutation of the i_j's solves the problem, but all the substitutions built with these permutations are equivalent.

   So, in this case, the unification problem has a single solution.

2. If $F \neq G$, only variables that are in the bodies of both terms, but not necessarily in the same position, can be exploited in further bindings. Let $\overline{i_r}$ and $\overline{j_r}$ be such that $s_{i_k} = t_{j_k}$ and $\{s_{i_1} \ldots s_{i_r}\} = \{\overline{s_p}\} \cap \{\overline{t_q}\}$, the solution substitution is

   $[\lambda\overline{s_p}.(H\ s_{i_1} \ldots s_{i_r})/F, \lambda\overline{t_q}.(H\ t_{j_1} \ldots t_{j_r})/G]$.

   In this case too, all permutations lead to equivalent substitutions, so that the unification problem has a single solution.

We call this operation *elimination of flexible-flexible pairs*.

So, we have proved that unification in L_λ is unitary.

Example 1.3.6

Terms $\lambda vwxyz.(U\ w\ x\ y\ z)$ *and* $\lambda vwxyz.(U\ v\ z\ y\ x)$ *are unified by* $[\lambda abcd.(V\ c)/U]$.

Variables v and w do not occur in the bodies of both terms. Variable x and z occur in the bodies of both terms, but in different positions. Variable y occurs in the bodies of both terms in the same position.

Example 1.3.7

Terms $\lambda vwxyz.(R\ w\ x\ y\ z)$ *and* $\lambda vwxyz.(S\ v\ z\ y\ x)$ *are unified by* $[\lambda abcd.(T\ b\ c\ d)/R, \lambda abcd.(T\ d\ c\ b)/S]$.

Variables v and w do not occur in the bodies of both terms. Other variables occur in the bodies of both terms, but sometimes in different positions. They are adjusted in the binding values.

1.3.4.2 Unification in L_λ is decidable

To show that λ-unification in L_λ is decidable, we observe that Huet's procedure, plus an occurrence-check, plus the elimination of flexible-flexible pairs (as defined in section 1.3.4.1), plus a special strategy, always terminates for problems in L_λ.

The strategy is to solve every flexible-rigid pair in a first phase, and then flexible-flexible pairs in a second phase. Remember that what Huet's procedure actually computes (without the elimination of flexible-flexible pairs) are pre-unifiers. By definition, pre-unified subproblems are solvable problems. So, the strategy is merely to pre-unify then solve.

Resolution of flexible-rigid pairs

The case of flexible-rigid pairs requires some attention because every step in MATCH replaces one unknown by possibly several (q in the notation of previous section) new unknowns (the H_i's), and also replaces one flexible-rigid pair by q simplified pairs $< \lambda\overline{x_n}.(H_i\ \overline{s_p}), \lambda\overline{x_n}.t_i >$. We have to express that the "size" of the problem decreases anyway.

We call *solved* an unknown that occurs as the head of a member of a flexible-rigid pair. Other unknowns are called *unsolved*. We call *depth of an occurrence* in a flexible pair the length of the path to that occurrence starting from the simplified pair. What decreases is the sum of the depths of all occurrences of unsolved unknowns.

The H_i's that come in the binding value do not count as unsolved unknowns because they occur as the heads of the new simplified pairs (so, they are solved). The β-reductions that become possible because of the substitution do not change anything about the depth of occurrences of unknowns because, in L_λ, β-reduction is restricted to β_0-reduction which amounts to renaming λ-variables[12]. Because of the occurrence-check, the depths of all unknowns (including unsolved unknowns) that occur in the t_i's is decreased by one.

Additional attention must be paid for dealing with degenerated cases in which there are no unsolved unknowns in the t_i's, or when q is 0. In these cases, the sum of the depths of all the occurrences of unsolved unknowns remain unchanged. Auxiliary measures are required such as the size of terms with no unsolved unknown.

To sum up, the vector whose first component is the sum of the depths of all occurrences of unsolved unknowns, and second component is the size of terms with no unsolved unknown, decreases in the lexicographic order at every application of MATCH if an occurrence-check is done. So, the first phase always terminates, yielding a set of flexible-flexible pairs (with a pre-unifier), or a failure.

Elimination of flexible-flexible pairs

When the first phase is terminated, every remaining pair is flexible-flexible, and their collection forms a solvable problem.

The case of elimination of flexible-flexible pairs is easy: it always suppresses one pair, never add any, and never change a flexible-flexible pair into a flexible-rigid pair.

So, the second phase always terminates, yielding an empty problem (a success-node).

We have used Huet's procedure for solving unification problems in L_λ. Miller proposes a more specialised (but we think more complex) algorithm which aims more directly to the goal. It does not create intermediary unknowns like the H_i's; this makes the termination proof easier.

Finally, note that Quian proposes a linear time and space algorithm for solving unification in L_λ [66], and that Pfenning applies a similar restriction to the terms of the Calculus of Construction [19] and proves that the corresponding unification problem is again decidable and unitary [65].

[12]This is the part of the reasoning that does not apply to the general case.

1.3.5 How to use L_λ?

Miller shows that every λProlog program can be mapped onto an equivalent L_λ program. Since λ-unification is certainly cheaper in L_λ than in λProlog, it seems to be a good idea to implement L_λ and then to map λProlog programs onto it. However, the very idea of a compile-time transformation misses the fact that many programs which are not in L_λ produce unification problems which are in L_λ.

Example 1.3.8
Higher-order predicates such as map_fun

```
type map_fun
        (A -> B) ->
        (list A) -> (list B) -> o.
map_fun  _Function  []  [].
map_fun  Function  [X|Xs]  [Function X|FXs] :-
        map_fun Function  Xs  FXs.
```

are not in L_λ (unknown Function *is applied to another unknown,* X*), but when used functionally (e.g. in mode* (map_fun + + -)*), and if* Function *and* X *are in L_λ, they only produce unification problems that are in L_λ.*

Finally, the mapping of λProlog programs onto L_λ programs does not work for polymorphic programs. The idea of the mapping is to write in L_λ what is reduction for every constant; the problem is that polymorphic constants stand for an infinite number of monomorphic constants. It may be that an extension of λProlog with more polymorphic types (e.g. terms of the second-order λ-calculus [7, 5] instead of simply typed λ-terms) can implement the mapping.

So, we prefer to implement the whole core of λProlog and to use the L_λ fragment as a special pattern to be recognised by procedure TRIV. It appears that the L_λ restriction encompasses every more *ad hoc* circumstance that TRIV can detect. To recognise flexible L_λ terms is especially interesting: it costs very little and gives a decision procedure where Huet's procedure can do nothing. In the following, we give several examples of λProlog programs. For many of them, we analyse whether they belong to L_λ or not.

1.4 Miscellaneous remarks

We present remarks on the pragmatics of λProlog: an interesting property of object terms, the complementary aspects of the terms extension and the formulas extension, and a comparison of implication (=>) and *assert/retract* as a means for modifying the program.

1.4.1 About combinators

Combinators are closed λ-terms. In λProlog, they may be non-pure combinators (i.e. combinators containing constants), but it does not matter.

It is an easy observation that if the two terms of a unification problem are combinators, then every derived subproblem is also made of combinators (because SIMPL propagates abstractions when deriving subproblems), and every binding value built up by the imitation and projection operations is also a combinator (because, in MATCH, neither imitation nor projection introduces free λ-variables).

It follows that the property of being a combinator is invariant through unification: unifying two combinators results in another combinator, and applying the unifier of two combinators to another combinator also results in a combinator.

λProlog ensures that every atomic formula is a closed λ-term because non-constant terms are either λ-variables explicitly quantified at the term level or unknowns implicitly quantified at the clause level or explicitly quantified at the goal level.

It follows that the terms of *all* unification problems produced when executing λProlog are combinators, and that every unknown stands for a combinator. This has consequences for the programming pragmatics and for the implementation. The second is studied in other parts of this report. We develop briefly the first in this section.

The pragmatical consequence is that every object term in a λProlog program is closed. I.e. predicate parameters and unknowns are combinators. So, there is no direct implementation of any statement dealing with non-closed terms. The only way to represent a non-closed term without introducing ambiguities is to replace the occurrences of free variables by universal variables. Implication can be used to attach properties to the universal variables. This is a very systematic pattern for dealing properly with abstractions and their bodies. See the clause implementing rule $\rightarrow_I$ in example 1.

Note that the second clause in example 1 is in L_λ: the only unknown, `E`, is applied to a single λ-variable, `x`.

Universal quantification is the source of many L_λ patterns because universal variables must be abstracted from terms for valuating unknowns that are introduced outside their scope.

1.4.2 The term language and the formula language

We have given a name, $\mathcal{CLP}(\lambda_\rightarrow)$, to the language in which the term domain is extended to simply typed λ-terms and the formulas are Horn formulas. However, this language is almost useless. We only took the care of naming it because the naming scheme already existed.

The language $\mathcal{CLP}(\lambda_\rightarrow)$ is almost useless because there cannot be any free λ-variables in object terms. λ-unification alone is not the proper tool for analysing or synthesising λ-terms (especially abstractions). Since there cannot be any free λ-variables in object terms, the only way for analysing or synthesising an abstraction is to apply the abstraction to some term. However, there is a problem in choosing the term. It should not clash with the subterms of the body of the abstraction, and should always be recognisable. This can be done with a severe coding discipline, or simply using universal quantification. However, when $\mathcal{CLP}(\lambda_\rightarrow)$ is enough, it is always possible to do a limited use of λProlog.

Example 1.4.1
The simple types program (see examples 1 and 1.1.1) can be rewritten in $\mathcal{CLP}(\lambda_\rightarrow)$.

```
/* Same declarations, plus ... */
type variable
       simple_type -> lambda_term.

/* Same first clause, plus ... */
has_type  (abstraction E)
          (arrow Alpha Beta) :-
       has_type  (E (variable Alpha))  Beta.
has_type  (variable Type)  Type.
```

The idea is that the relation between a term and its type is functional, and that we need not distinguishing between variables that have the same type. So, typing statements can be stored in situ.

The status of constant `variable` *is ambiguous because it does not correspond to a construct of the modelled language, but it cannot be distinguished from the other constructs. The $\mathcal{CLP}(\lambda_\rightarrow)$ implementation must be used in mode* `(has_type + ?)`*, otherwise it may produce a term with undistinguishable constructs* `variable`.

The language with hereditary Harrop formulas and first-order terms is more useful, but the encoding of its formulas into its terms is difficult. On the opposite, the language of λ-abstractions gives a natural notation for every kind of quantification.

A tag added to an abstraction (e.g. `pi` or `sigma`) indicates that some semantics is added. The semantics of `pi` and `sigma` is implemented by the λProlog system, whereas the semantics of user-defined quantifiers must be implemented by the user itself. In examples 1.1.1 and 1, constant **abstraction** is a tag which gives the meaning of an object level abstraction to an abstraction. The semantics, with respect to types, is given in predicate **has_type**. See also how predicate **setof** is considered as a quantifier constructor in section 4.1.1.3.

So, the extensions of both the terms domain and the formulas domain are simultaneously needed.

1.4.3 Implication vs. *assert/retract*

Standard Prolog proposes built-in predicates **assert** and **retract** for modifying the program. Implication in goals has also the effect of modifying the program, but there is no way to express one with the other.

Predicate **assert** augments a program by a clause that is conformed on the model of a term but that has a new universal quantifier for every unknown that occurs in it. On the opposite, implication augments a program with a term that is a clause and it does not invent any quantification; it only interprets the quantifications that are explicit. So, implication may introduce free logical variables in programs.

Example 1.4.2
In Standard Prolog, goal

```
        assert  (p U V)
```

adds clause[13]

```
pi U\(pi V\( p  U  V ))
```

In λProlog, implication goal

```
        p  U  V => G
```

adds clause[14]

```
p  U  V .
```

Where variables U *and* V *stand for terms that are chosen by the proof process. This cannot be done with predicate* **assert** *because the resulting clause is not even a Standard Prolog clause. To achieve the same quantification effect in λProlog as in Standard Prolog, one has to execute goal*

```
        (   pi U\(pi V\( p  U  V ))
        =>  G
        ).
```

The Standard Prolog and λProlog ways of augmenting the program are also different as for the lifetime of the new clause. In Standard Prolog, the new clause remains in the program until it is explicitly retracted. In λProlog, the new clause remains in the program for the proof of the conclusion of the implication of which it is the premise (goal G in example 1.4.2).

[13]The usually implicit quantification is made explicit for the purpose of explanation.

[14]In this clause, variables U and V are *free* in the clause but existentially bound in the proof.

In this particular occurrence, one must not apply the syntactic convention saying that variables free in a clause are implicitly universally quantified.

In λProlog like in Standard Prolog, predicate names have a global scope and names of unknowns have a local (to a clause) scope. Moreover, terms have no name in general. Since implied clauses are not disjoint from the proof context (they may share terms) implication is a way to give a global name to a term. This is an important pragmatic feature of implication. In Standard Prolog, the only way to have two identical terms used in two different occurrences is to maintain a continuous chain of parameter passing between the two occurrences. In λProlog, implication can be used to this end because it gives a global name to a term.

Example 1.4.3
Predicates **append** *and* **reverse** *can be defined in the following way:*

```
type append
        (list A) -> (list A) -> (list A) -> o.
type app
        (list A) -> (list A) -> o.
append  X  Y  Z :-
        (   app  []  Y
        =>  app  X  Z
        ).
app  [E|X]  [E|Z] :-
        app  X  Z.

type reverse
        (list A) -> (list A) -> o.
type rev
        (list A) -> (list A) -> o.
reverse  X  Y :-
        (   rev  []  Y
        =>  rev  X  []
        ).
rev  [A|X]  Z :-
        rev  X  [A|Z].
```

In both predicates append *and* **reverse**, *unknown* Y *is common to the new clause and the proof context. In both cases, unknown* Y *is given a kind of global name (*(app []) *or* (rev [])*) for the duration of the proof of a goal (*(app X Z) *or* (rev X [])*). Finally, in both cases, unknown* Y *disappears from the goal-statement when the implication goal is executed and re-appears later when solving a goal. In the meantime, unknown* Y *was only present in the program.*

1.5 A running example (continued): from d-lists to f-lists

This section continues the "running example" presented in the introduction. It insists on using the λ-terms as data-structures and sheds some light on the connection between unification and universal quantification in goals.

Function-lists can yield the same improvement as difference-lists while allowing a list to be a left concatenand more than once. This is the logical advantage of function-lists over difference-lists. It is up to the implementor to make them "non-deceptive". For instance, in contexts in which difference-lists are correct implementations of lists, using function-lists instead should yield the same complexity.

Hughes presents a similar list representation in the functional programming framework [33].

1.5.1 Function-lists

A list is represented by the function that left-concatenates the Prolog representation of the list to its argument. For instance, the empty list is represented by `z\z` and $cons(1, cons(2, cons(3, nil)))$ is represented by `z\[1,2,3|z]`. The function-list representation is unique up to λ-equivalence. A function-list has type `(list A) -> (list A)` if its elements have type `A`. Though there are no way in λProlog for declaring a type function `flist`, type `(list A) -> (list A)` is abbreviated to `(flist A)` throughout this report. Type constructor `flist` has kind `type -> type`. *One would like to write*

```
defkind flist
        A\((list A) -> (list A)).
```

but it is not legal in λProlog.

To concatenate two lists is to compose the functions that represent them. The concatenation predicates are:

```
type fappend
        (flist A) ->
        (flist A) ->
        (flist A) -> o.
fappend  L  R  z\(L (R z)).

type fappend3
        (flist A) ->
        (flist A) ->
        (flist A) ->
        (flist A) -> o.
fappend3  L  M  R  z\(L (M (R z))).
```

Note that these clauses are not in L_λ. But, when used in functional modes (i.e. `(fappend + + -)` and `(fappend3 + + + -)`), they only produce trivial unification problems.

Unlike predicates **dappend** and **dappend3**, predicates **fappend** and **fappend3** operate in all modes because equality of function-lists is completely encompassed by higher-order unification. In mode `(fappend3 - - - +)`, the backtracking implementation of unification enumerates the different possible splits of the fourth parameter. Predicate **fappend** and **fappend3** are not destructive because of the semantics of β-reduction. So, the following predicates work as expected.

```
type ftwice
        (flist A) -> (flist A) -> o.
ftwice  L  LL :-
        fappend  L  L  LL.

type common_prefix
        (flist A) ->
        (flist A) ->
        (flist A) -> o.
common_prefix  P  L1  L2 :-
        fappend  P  _  L1,
        fappend  P  _  L2.
```

The check for membership can easily be derived from predicate **fappend**.

```
type fmember
        A -> (flist A) -> o.
fmember  E  z\(_ [E|(_ z)]).
```

In the same fashion, predicate **fselect** relates a list, one of its elements, and the list of the other elements.

```
type fselect
        A -> (flist A) -> (flist A) -> o.
fselect  E  z\(B [E|(A z)])  z\(B (A z)).
```

The naive reverse predicate is:

```
type fnrev
        (flist A) -> (flist A) -> o.
fnrev  z\z  z\z.
fnrev  z\[A|(L z)]  z\(RL [A|z]) :-
        fnrev  L  RL.
```

The first clause is obvious since `z\z` represents the empty list. The second clause uses higher-order unification to split a list and construct another.

As we did with difference-lists, function-lists can be used to produce an inversion predicate that operates on Prolog lists but uses function-lists internally.

```
type frevl
        (list A) -> (flist A) -> o.
frevl  []  z\z.
frevl  [A|L]  z\(RL [A|z]) :-
        frevl  L  RL.

type frev
        (list A) -> (list A) -> o.
frev  L  (RL []) :-
        frevl  L  RL.
```

1.5.2 Function-lists and universal quantification

The transformation from function-lists into Prolog lists is trivial and is not destructive.

```
type flist2list
        (flist A) -> (list A) -> o.
flist2list  F  (F []).
```

In spite of (because of) λ-unification, predicate `flist2list` is not totally symmetrical. It cannot be used to transform a Prolog list into a function-list because there are other solutions than the intended one, and there is no way to "separate the wheat from the chaff" *a posteriori*. For instance, let `[[1]]` be a Prolog list the only element of which is a list. The solutions to `< [[1]], (F []) >` are `[z\[[1]] / F]` and `[z\[[1]|z] / F]`. The intended solution is the second, first solution is "chaff". The non-intended solution is produced by an excessive use of the imitation rule.

Note that substitutions `[z\[[1|z]] / F]` and `[z\[[1|z]|z] / F]`, which are produced by projection, are not solutions because they violate the type assignment. In this problem, unknown `F` has type

```
(list (list int)) -> (list (list int)) ,
```

but first substitution gives it type

```
(list int) -> (list (list int)) ,
```

and second substitution cannot even be typed because the first occurrence of variable `z` has type `(list int)` while its second occurrence has type `(list (list int))`.

The problem is basically a logical problem. Predicate `flist2list` is not a correct implementation of the relation between Prolog lists and function-lists, which is

> *To append a Prolog list to a given list is to apply the corresponding function-list to the same list, for* every *given list.*

Predicate `flist2list` implements the relation only when the given list is the empty list. We have to design another predicate for doing correctly the job for *every* given list.

We propose two versions[15]:

```
type list2flist
        (list A) -> (flist A) -> o.

#       ifdef first_version
list2flist  L  FL :-
        pi list\
        (  append  L  list  (FL list)
        ).
#       else  second_version
list2flist  []  z\z.
list2flist  [E|L]  z\[E|(FL z)] :-
        list2flist  L  FL.
#       endif
```

[15]Notation `# ifdef ... # else ... # endif` is borrowed from the C preprocessor. It expresses conditional reading of programs.

The second version uses cautiously the structure of function-lists. The first version is more interesting in that it reproduces directly the relation between Prolog lists and function-lists. It makes a critical use of the universal quantification. In the example above, `pi` introduces the universally quantified constant `list`, which cannot be captured by imitation.

Predicate `flist2list` is not in L_λ. The two versions of predicate `list2flist` are in L_λ. The first version satisfies the condition on quantification nesting that is given in section 1.3.3.

1.5.3 Function-lists and combinators

A concatenation combinator can be written and used autonomously. It is the function composition combinator, `l1\l2\z\(l1 (l2 z))`. In the same vein, the nil combinator is `z\z`.

Statements about the properties of lists and concatenation can be expressed in the language itself, and the empty list and concatenation combinators can be generated automatically.

```
type monoid
        (   (flist A) ->
            (flist A) ->
            (flist A)
        ) ->
        (flist A) -> o.
monoid  APPEND  NIL :-
        pi l1\(pi l2\(pi l3\
        (      (APPEND (APPEND l1 l2) l3)
             = (APPEND l1 (APPEND l2 l3))
        ))),
        pi l\
        (   (APPEND l NIL) = l ,
            (APPEND NIL l) = l
        ),
        pi e\
        (   (APPEND z\[e|z]) = l\z\[e|(l z)]
        ).
```

The current state of the art of λProlog implementation makes this more a curiosity than a tool. In mode `(monoid + +)`, the predicate `monoid` can check that two combinators have the required properties. In mode `(monoid - -)`, the same predicate finds solution

```
[x\y\z\(x (y z)) / APPEND, z\z / NIL] .
```

The fourth literal is necessary for describing the relation between `APPEND` and `'.'`. When the fourth literal is omitted, another solution where `x` and `y` are permuted is possible. In both cases, the unification procedure may enter a loop after finding the solution(s).

Note that the proof of `(monoid APPEND NIL)` makes use of unification delay. The first literal produces a flexible-flexible pair which becomes a constraint. The other literals awake the constraint every time a binding is found for `APPEND`.

Combinators `APPEND`, `NIL` and some others can be used to write function-lists predicates more safely[16].

[16]Notation `#define` is borrowed from the C preprocessor. It expresses definition of named strings. We use it only for naming strings that can be analysed as closed ground terms.

```
#define APPEND     x\y\z\(x (y z))
#define NIL        x\x
#define UNIT       x\z\[x|z]

fappend  L  R  (APPEND L R).

fappend3  L  M  R  (APPEND L (APPEND M R)).

ftwice  L  (APPEND L L).

fnrev  NIL  NIL.
fnrev  (APPEND (UNIT A) L)
       (APPEND RL (UNIT A)) :-
        fnrev  L  RL.

fmember  E  (APPEND _ (APPEND (UNIT E) _)).

fselect  E
         (APPEND B (APPEND (UNIT E) A))
         (APPEND B A).
```

It makes programs safer and more readable, and it costs nothing at run-time because programs are reduced at compile-time. In the same way, concrete data-structures and notational conventions can be hidden in combinators. Figure 4.3 in section 4.1.1.5 shows another example of this trick.

Chapter 2

Technical Choices

Our implementation of λProlog relies upon two technical choices. The first, to use MALI for representing the search-state, is a condition of the memory efficiency of our system. The second, to compile to C, is incidental. It is only an easy way to get the system integration ability we look for.

Examples of programming with MALI can be found in section 3.1.

2.1 MALI

MALIv06 is a *software tool* designed to provide a *user program* with the *memory management* of a *search-stack*. The intended application domain is the implementation of *logic programming systems*.

The principles implemented in MALIv06 are fully exposed in a tutorial and reference manual from which this section is an edited excerpt [69]. In the following, MALIv06 stands for this particular implementation of the principles, and MALI stands for the principles themselves. MALI (Mémoire Adaptée aux Langages Indéterministes — memory for non-deterministic languages) can be specified as the abstract data type *stack of mutable first-order terms*.

We present in the following the domains of MALIv06's data-structures, and the great lines of the memory management.

2.1.1 The domains of MALIv06

MALIv06 allows to store complex data-structures called *terms*, to modify them in a controlled way, and to save them on a *term-stack* for protecting them against modifications.

2.1.1.1 Commands, operations, types, constants

As a memory, MALIv06 has a state and *commands* exist to either alter or consult its state. The semantics of MALIv06 is mainly the semantics of its commands. Commands are an enrichment of the read and write commands of a conventional memory.

Companions to the commands are the *operations*. Their behaviour does not depend on the state of MALIv06 and they do not alter it. They hide the structure of MALIv06's internal representations while giving access to them.

MALIv06 introduces new computation domains to describe what can be stored in its memory. Some of the domains are public —i.e. they are part of the user interface—, others are private. Private domains are described in the documentation for the purpose of specification and explanation. When a domain is public, a *type* is defined to allow the user to store values of the domain and compute with them. Constant values and functions are declared for some public domains.

2.1.1.2 Terms and names

The *terms* are private values stored in MALIv06. They only serve as a semantic tool. The *names* are public values related to terms by a partial function called the *designation* (see section 2.1.1.3). The user of MALIv06 never gets any term value, but only names stored in *cells*. Cells have type T_CELL.

Cells are the storage unit of MALIv06. The memory of MALIv06 is an aggregate of cells which can be pointed to by a user program.

A term has a *nature* specifying which commands may apply to it, and a *sort* for an elementary typing. Domains of nature and sort are public. Cells have an *indicator* field for storing nature and sort, and an *information* field which is interpreted according to the indicator. Natures, sorts, and informations have types T_NAT, T_SORT, and T_INFO.

Some terms, called *compound* terms, have *subterms* which are also terms.

All compound terms plus some others exist in a mutable and a non-mutable form. Mutable terms are called *muterms*. They have the same structure as non-mutable terms, but their usual destiny is to be replaced by another term (of any nature and sort). In this respect, they are similar to Prolog logical variables. Commands exist to perform the replacement.

The terms used in this report have the natures listed below. For every nature, we give a notation for terms of this nature. This notation is not used for communicating with MALIv06; it is only used in writing comments.

Atoms They have an information associated to them which cannot be interpreted by MALIv06. It is an arbitrary value and it is only stored unaltered. The information size is restricted to what can be stored in the information field of a cell.

In the following, an atom of sort S and value V is noted (at S V).

Binary compounds They are similar to the *cons* of Lisp. They have two subterms. There are also nullary, unitary, and ternary compounds.

In the following, a binary compound (mutable or not) of sort S and subterms T1 and T2 is noted[1] ([m]c2 S T1 T2).

Nullary Compounds They are similar to the *nil* of Lisp. They are degenerate compounds since they have no subterms. Mutable nullary compounds are similar to Prolog logical variables. All other muterms can be thought of as being decorated logical variables.

In the following, a nullary compound (mutable or not) of sort S is noted ([m]c0 S).

Tuples They are n-ary compounds. They have n ($n > 0$) subterms. They offer a more compact representation than a combination of *n-1* binary compounds. Furthermore, they offer direct access to their subterms. The maximum size of tuples is discussed in the tutorial and the reference manual.

In the following, a n-ary compound (mutable or not) of sort S and N subterms T1 to Tn is noted ([m]tu S N T1 ... Tn).

Levels An important component of MALIv06 is the *term-stack*. Levels are substacks of the term-stack. They are ordinary terms and can be subterms of any compound term. Levels have a subterm called a *root*. It is the term which is saved on the term-stack.

Conventionally, the term-stack grows upwards, so that a level is called lower than another if it is a substack of the other. Symmetrically, a level is called higher than another if the other is a substack of the level. Commands exist to replace levels by other lower levels.

There is only one term-stack at a time, and every level is one of its substack. So, when the term-stack is not empty, there always exist a highest level (called the *top level*) and a lowest level (called the *bottom level*).

In the following, a level of sort S, root R, and next level N is noted (le S R N).

There are no other term replacement operations than the replacement of muterms by terms and the replacement of levels by lower levels.

The terms of MALIv06 can be considered as *infinite rational terms* because they may be their own subterm, but they may only have a finite number of different subterms.

The notation of MALIv06 terms is extended to represent sharing of subterm. A subterm t can be labelled l using the following form: l@t. Every other occurrence of the label stands for the subterm. A term may have several labels (e.g. through substitution). Another way of seeing the terms of MALIv06 is as graphs.

[1] The square bracketed m ([m]) indicates the mutable variant of the nature.

2.1.1.3 The designation

The relation between names and terms is specified by a partial function called the *designation*. The reference manual of MALIv06 intends to expose precisely how the designation evolves.

The designation is an almost constant function from names to terms. We say that a name designates a term. By extension, names are said to have the nature and sort of the term they designate.

Atoms are their own names. The information associated to them is the content of the information field of their names (themselves).

Replacing a muterm by a term and a level by a lower one causes the only non-trivial modification of the designation. Other modifications are term creation and term suppression. They are trivial since they only modify the domain of the designation. Term creation is always explicit, whereas term suppression is always implicit.

Term replacement always makes several names designate the same term.

One says that a name designating a term without the help of a term replacement is a *direct name*.

It is always possible to find a direct name designating the same term as a given non-direct name.

A command exists for this purpose. Since the arguments of many commands must be direct names, non-direct names may cause run-time errors. It is up to the user to determine where non-direct names can pop out, and to make them direct when necessary.

Terms can be *saved* on the term-stack to be partly isolated from the modifications of the designation. There are two kinds of term replacement which act differently on the saved terms. The first is the *substitution* of a term to a muterm. It has no effect on the saved terms. Once popped off the term-stack, a term shows no sign of substitutions done after it was saved. The second kind of replacement is the *assignment* of a term to a muterm, or of a level to a higher level. It affects the saved terms as well as the designation. So, a term popped off the term-stack keeps the effect of assignments.

2.1.1.4 The objects

Terms are represented by contiguous subaggregates of MALIv06's memory called *objects*. The way the cells are aggregated in objects is of no purpose for the user.

Objects are the allocation unit of MALIv06.

The domain of objects is private. It serves only a specification purpose. The domain of *references* to objects is public. They can be used and stored by a user program. There are operations that apply only to cells aggregated in objects. References to objects have type T_ROBJ.

Objects can be seen as the descriptors of the terms. Their component cells are the fields of the descriptors. The way objects aggregate cells is hidden. However, given the reference of an object representing a term of some nature, *selection operations* return a pointer to a particular cell of the aggregate. The cells that can

```
        ------------------------ Read & Make commands ---------------------->


                 designates               is represented by      is referenced by
  name1 ------------------------->    --------------->       -------------->
                                  term1                 object1                reference
  name2 <------- term2 <----------    <---------------       <--------------
        has name       has subterm           represents             references


        <----------------------- Selection operations ----------------------
```

Figure 2.1: Names, terms, objects and references

be pointed to in that way correspond to subterms of the term. They contain the name of the subterm they correspond to.

2.1.1.5 Making and reading objects

Making and reading objects follow two rules:

> *All terms are built one compound at a time*

and

> *All terms are read one compound at a time.*

Make commands exist for building a term of any given nature and sort. They create a suitable object for representing its constructor, and they return

1. the name of the new term in a cell chosen by the user, and
2. the reference of the object.

The cell chosen by the user is often selected (using a selection operation) in an object created earlier, while the object reference serves for building subterms.

To read a term, one must know a cell where the name of the term is stored. The cell can be either in the representation of a term or elsewhere. The result of reading a term is the reference of the object in which the cells containing the names of the subterms are aggregated.

Note that make and read commands are totally symmetrical. Figure 2.1 summarises the relations between names, terms and subterms, objects and references, and selection operations and read and make commands. Recall that only names and object references are public.

An analogy can be drawn between term naming in MALIv06 and file naming in an operating system. A file name designates a file in a file system. To actually do something with a file, it must be opened in a mode suitable for the intended use (say, read or write). To open a file yields a file descriptor which must be used for every access to the file. Noting that file directories are also files, it appears that terms of MALIv06 are somewhat like files[2]. The names, objects, and read and make commands of MALIv06 correspond to file names, file descriptors, and to opening a file in read or write mode. A file descriptor is a structured data; selection operations implement field selection on objects.

[2]The analogy flounders as soon as rational terms are considered.

2.1.2 Memory management

2.1.2.1 Usefulness logic

The principles of MALI result from a research on the implementation of logic programming systems which focused on memory management rather than speed. So, MALI provides an efficient solution to the space problem, but brings no solution to the time problem. The packaging of MALIv06 is designed to hinder the least possible any effort to yield speed efficiency.

Logic programming systems require management of their program space because programs are modifiable. They also require management of the space in which the state of the search process is stored. The memory management of the program space is dependent on language issues that are not universally settled. The memory management of the search process appears to obey some rules and has a general solution which can be wired in a garbage collector. The main rule is that

> *The data-structures implementing the search must be interpreted by the garbage collector.*

MALIv06 is a packaging of this rule which is as much as possible independent from any logic programming system.

We call the interpretation of the data-structures for garbage collection purpose the *usefulness logic* of a system A survey paper by Bekkers, Ridoux and Ungaro describes the emergence of this notion in logic programming [10].

In logic programming systems, a search results in the binding of some logical variable to some values. The search algorithm defines a notion of search node and search transition between nodes. Transitions produce new bindings, and the bindings produced along the path leading to a given search node form a *binding environment*. It happens that a search node may be the source (by the transition rules) of several other search nodes, which in turn, etc. So, an implementation of a logic programming system has to represent soundly and as efficiently as possible a collection of search nodes and their binding environments. Note that it is common practice to use a given data-structure in the representation of several binding environments (sharing). So, the binding environments considered as graphs may share vertices and edges.

MALI's usefulness logic specifies that

> *The run-time data-structures representing binding environments must be interpreted as vertices of as many graphs as there are environments. Those that have at least one connection to some root using edges of a single graph are the only useful ones.*

As a corollary, a data-structure that is either not connected at all to any root or that cannot be connected without using the edges of several graphs is useless. In the latter case, Lisp usefulness logic would say that the data-structure is useful.

MALI's muterms, levels and compound terms correspond to Prolog logical variables, search-stacks, and terms and goal-statements. So, all what is said about MALI can be transposed to Prolog.

2.1.2.2 Two new features

Muterms introduce two new features distinguishing MALI's usefulness logic. The two features can be described with respect to the two roles that an object representing a muterm plays alternatively. First, when no term is substituted to the muterm, the object represents only the muterm, it is a place-holder. Second, when a term is substituted to the muterm, the object represents the substitution by keeping a name of the term, but it is ready to represent the muterm again if it is popped off the term-stack.

A complete implementation of MALI's usefulness logic must detect that an object actually plays only one role and may be replaced by a cheaper representation of the remaining role. Note that if an object does not play any role, it can be discarded. Since an object may play no role as a consequence of the loss of one of the roles of another object, a conventional usefulness logic is not certain to recognise every object that plays no role, without speaking of recognising objects that play only one role.

Early reset

The first feature is called *early reset* or *binding shunting.* It corresponds to the case in which the second role is missing. Assume an object representing a substituted muterm that is connected to some roots only in binding environments in which no term is substituted to the muterm. Such an object actually plays only the first role and *the substitution can be undone.* The potential gain is the representation of the substitution value.

"Early reset" gets its name from the fact that a substitution is undone without waiting for backtracking. Note that a substitution value can be any arbitrary large structure. So, the gain of early reset is arbitrary.

Muterm shunting

The second feature is called *muterm shunting* or *variable shunting* because of the relation between muterms and logical variables. It corresponds to the case in which the first role is missing. Muterm shunting can either be global or local, according to the scope of its effect.

Global muterm shunting is based on the object representing a given muterm. Consider again an object representing a substituted muterm. Assume it is connected to some roots only in binding environments in which a term is substituted to the muterm. Such an object actually plays only the second role and need not be ready to represent the muterm again. Then, *every occurrence of the name of the muterm can be replaced by the name of the term.* The potential gain is the representation of both the muterm and its subterms.

Local muterm shunting is based on the occurrences of the direct name of a muterm. It does not improve memory allocation, but only shortens substitution chains. If an occurrence of the direct name of a muterm is connected to some roots only in binding environments in which a term is substituted to the muterm, then *this occurrence of the name of the muterm can be replaced by a name of the term.*

It is important to understand how the two kinds of variable shunting are different. The first operates on the muterm itself and all its occurrences. The second operates on particular occurrences of a muterm. The first may induce a memory gain, whereas the second cannot. Both shorten substitution chains.

Assuming that substitutions are implemented by pointers, what muterm shunting does is to collapse pointers; the usefulness logic of functional programming system has no ground for such an action.

In the case of Prolog, muterms are only used for representing logical variables: they are nullary muterms. So, the gain of muterm shunting seems to be a constant amount. However, as logical variables get more complex (types, constraints) their representation becomes arbitrary large. So, the gain of not representing them becomes also arbitrary[3]. Note also that as logic programming systems get more complex, muterms are also used for more purposes.

2.1.2.3 Stack-like memory management

In addition to a garbage collector, MALIv06 benefits from term-stack management to recover memory for free. Popping the stack actually frees memory, so that if the garbage collector is not used, MALIv06 behaves as well (as badly) as a Prolog system with no garbage collection.

A less extremist consequence, but more interesting, is that short lived objects at the top of the stack are

[3] In usual implementations of the WAM (Warren Abstract Machine [75, 4]), some logical variable have no proper representation: they are represented by a slot in one of the terms in which they have an occurrence. It seems that muterm shunting saves no memory for these logical variables.

It is false: even with the representation as a slot, it may be that a slot is only used for representing a logical variable and never as a component of a compound term. Muterm shunting helps recover it.

Moreover, not every logical variable can be represented this way (*unsafe variables* need a proper representation), and it cannot be used either for representing complex logical variables (they need more room than a single slot). On the opposite, muterm shunting is a general device.

discarded at no cost. The cost of garbage-collection is paid by long lived objects.

2.2 The Prolog-to-C translation

The translation from Prolog to an imperative language is motivated by software engineering objectives, and the facility of inheriting its back-end from an already existing compiler.

We present an execution model that merges smoothly the use of MALI and the translation to an imperative language.

2.2.1 Objectives

Our objectives for compiling λProlog include the usual efficiency obsession: to make λProlog reasonably efficient, and above all non-deceptive. Its attractive features should be truly usable. They also include a software engineering concern: we want to revisit the integration of Prolog in a host system.

Several reasons make one feel unsatisfied with the current style of Prolog integration.

1. It is often the case that Prolog is only accessible through the use of a specific programming environment: the so-called "supervisor". This makes the use of Prolog independent of the host system, but it insulates Prolog applications from non-Prolog ones.

   However, it is current practice to compose programs written in different languages with the help of a shell language. For example. the pipe construction (|) of some shell language composes programs by connecting their standard inputs and outputs and does not matter for the languages they are written in[4].

2. Numerous attempts have been made to endow Prolog with a capability for modular programming. Every attempt we know about claims to give a Prolog semantics to modularity [47, 48]. We believe that it misses the part of the point which is to link at the system level modules that were designed independently and in languages we do not want to know about.

   We want to make it possible to compose, share and reuse program texts (among them, some written in Prolog) by the exclusive use of symbols. To this effect, one must give the scoping laws of the different declarations of a Prolog text, and rely on the link-editor of the host system.

   This does not preclude a more Prologish way of composing modules inside the Prolog parts of a program.

So, our objective is to implement Prolog, whichever variant is chosen, so that Prolog applications are as well integrated in the host system as applications written in other languages. Furthermore, every standard behaviour of the host system applications must be reproduced: standard input/output/error ports, exit codes, etc.

To achieve this at low cost, we translate Prolog source programs into source programs of a language which already enjoys the integration. This is not enough to have Prolog inherit the same integration capability. The translation must maintain a correspondence between Prolog symbols and the target language symbols so that link editing at the target level performs the expected effect at the Prolog level.

A more efficient way of achieving the same effect would be to translate to some portable compiler back-end format. It would save some compilation time and execution time, while preserving machine independence. But, it is easier to use a real programming language.

2.2.2 Model of execution

We use a new model for the execution of Prolog programs. It has been first applied to Standard Prolog, and then to λProlog. We describe this model for Standard Prolog.

2.2.2.1 Standard Prolog

Standard Prolog programs are translated into C programs [37] that use MALI for representing the search state. The search state is made of a success continuation (often called *goal-statement*) and a failure continuation (often called *search-stack*). Both continuations are coded as MALIv06 terms. The success continuation uses compound terms, atoms, and muterms for representing goals and terms constructors, symbols and constants, and unknowns and mutable structures (β-redex, etc). The failure continuation is mapped onto the term-stack.

The mapping of continuations onto MALIv06 is hidden in the definition of a specialised intermediate machine built on top of MALIv06. Like MALIv06, its interface is mainly a set of commands and operations.

The translation scheme is to perform a top-down analysis of Standard Prolog programs, and to associate to every structure and symbol a declaration or a statement. From the top to the bottom of the analysis, we meet predicates, clauses, heads, bodies, and terms. A predicate translates into a symbol definition and a function implementing the part of the search procedure that is related to the predicate. The function performs side-effects on both continuations. A clause translates into command calls implementing the search strategy (depth-first search), the head and the body. Every head atom translates into a sequence of command calls implementing a specialisation of unification suitable for this head. Body goals translate into a sequence of command calls implementing the proof strategy (left-to-right selection of goals). Finally, we have head structures (compound terms, constants, unknowns) which translate into command calls implementing unification specialised for them, and body structures which translate to command calls for creating new goals and adding them to the success continuation.

[4]Some Prolog systems offer a mode for generating standalone applications.

The C language serves as a glue for putting together the command calls to the specialised intermediate machine. The call-stack of C is used neither for implementing the traversal of the search-tree, nor the traversal of the proof-trees. It would kill the benefit of using MALI because MALI has to know about the search-stack but cannot go through the C call-stack. So, the functions implementing search for the predicates never call each other. They are called by, and they return to, a unique function called the *motor*. It is an extremely simplified scheduler or interpreter.

C is not the best language to serve as a glue. It lacks the ability to consider as a data a control point other than a function pointer. Language PL/1 extends this ability to labels and entries. However, the wide availability of C makes it to be chosen despite this limitation.

We have slightly extended the syntax of Prolog for merging Prolog and C declarations in a single file, and writing clause bodies in C. It is chiefly useful for implementing the built-in predicates or for interfacing host-system libraries.

2.2.2.2 Extension of the execution model for λProlog

To extend the model for handling λProlog, one has to add two new continuations: one for handling the signature, the other for the program [12, 14]. Remember that both the signature and the program may change during a computation. These two new continuations are represented as MALIv06 terms (see details in sections 3.3 and 3.2.3.1).

Because of the non-determinism of λ-unification, it is not possible to merely substitute the λ-unification procedure to the first-order one. We follow the tradition of a depth-first search of the unification tree so that the non-determinism of unification merges smoothly in the non-determinism of the proof-search. It merges so smoothly that procedure MATCH (see figure 1.2) is a λProlog predicate that non-deterministically calls the imitation and projection procedures through clause bodies written in C.

2.2.2.3 Management of Prolog symbols

To obtain that link editing at the target language level implements Prolog link editing, one must make sure that every linkable symbol of Prolog is mapped onto a linkable symbol of the target language.

We achieve this by translating every symbol into a pointer to a structure that contains the description of the symbol (external representation, arity, a type reconstruction function, a search function for predicates symbols, etc). The set of all this structures can be seen as a symbol table.

Chapter 3

Compiling λProlog

A bunch of new problems comes with the extension to higher-order terms: terms are typed, they must be considered modulo the λ-equivalence relation, two quantifiers (universal, pi, and existential, sigma) and a new logical connective (implication, =>) are introduced, and unification problems can have several solutions and they can be delayed.

We have to design an execution scheme for λProlog and then study how it can be specialised according to recognisable source patterns. The new scheme should be a mix of three different computing technologies. First, the Prolog technology (goal and clause selection, substitution, backtracking) is used because λProlog is an almost conservative extension of Standard Prolog[1]. Second, the λ-calculus technology ($\alpha\beta\eta$-equivalence) must cope with the extension of the term domain to simply-typed λ-terms. Finally, the uniform proof technology for hereditary Harrop formulas ($\supset_R$, $\forall_R$) has to cope with the extension of the clause form.

All three technologies meet in unification. Unification is the place which is really different from what it is in Standard Prolog. However, part of the compilation effort is to recognise patterns for which a mild alteration of first-order unification does the job.

The Prolog technology is implemented by a specialised intermediate machine called the *MPPM* ("MALIv06 Pedagogical Prolog Machine"). We first introduce it and its implementation with MALIv06, and then the new problems. Implementing the two other technologies is required to extend the MPPM. We present the data-structures and only give hints on the operations.

3.1 The MALIv06 Pedagogical Prolog Machine: MPPM

The general architecture [13] of a system programmed with MALIv06 is the following:

- The bottom layer, MALIv06, offers a general data-type for representing application oriented terms. It also offers an efficient memory management.

  It does not own any memory; the memory it manages is provided by higher layers.
- The virtual machine layer defines how application oriented terms are represented as MALIv06 terms. It also implements the operations defined on these terms (resolution, unification, reduction). It may also define the great line of the memory policy (what are the actual memory resources, how large are they, are they paged, etc).

  It says nothing of what is a program.
- The program layer may be made of interpreted code or of executable code. We choose the second in which the primitives of the virtual machine layer are called from a target language (in our case C).

The MALIv06 tutorial completely describes the MPPM [69]. The content of this section is an edited excerpt of the tutorial. It describes the parts of the MPPM that are the most important for understanding the execution scheme. An important part missing in this section describes the control of MALIv06's memory management (the *memory policy*).

MALIv06 is written in C and produces C definitions. So, the preferred user language is C. However, the use of C is purely incidental, and we try not to use too many C tricks.

The MPPM has to deal with the mapping of Prolog domains, control, and unification onto MALIv06. In the following, commands with two-letter prefixes (MK, RD, ...) are MALIv06 commands, and commands with longer prefixes (MAKE, READ, ...) are MPPM commands.

3.1.1 Prolog domains

An important task in the implementation of a logic programming system with MALIv06 is to devise the mapping of the application domains onto the terms of MALIv06. As a mapping is chosen, we recommand to design a set of application oriented commands. They should be a specialisation of MALIv06 commands.

3.1.1.1 Natures and sorts

The domains of the application (e.g. Prolog terms) must be mapped onto the terms of MALIv06. Natures and sorts must be chosen and combined properly to have an

[1] It is *almost* conservative because definitional genericity (see section 3.2.1.1) makes some Standard Prolog programs definitely ill-typed (e.g. a flattening predicate for unbounded nestings of lists: program 9.1a page 136, and its variants, in *The Art of Prolog* [72]).

```
#define SORT_INT                          MK_SORT( 1984 )
#define SORT_UNK                          MK_SORT( 314159 )
#define SORT_FUNC                         MK_SORT( 1515 )
#define SORT_SYMB                         MK_SORT( 464 )
#define SORT_GOAL                         MK_SORT( 13 )

#define INDIC_INT                         MK_INDIC( N_ATOM,   SORT_INT )
#define INDIC_UNK                         MK_INDIC( N_MCONSO, SORT_UNK )
#define INDIC_FUNC                        MK_INDIC( N_TUPLE,  SORT_FUNC )
#define INDIC_SYMB                        MK_INDIC( N_ATOM,   SORT_SYMB )
#define INDIC_GOAL                        MK_INDIC( N_TUPLE,  SORT_GOAL )

#define FUNCTOR_OF( objfunc )             TUPLE_SELECT_ITH( objfunc, 1 )
#define SUBTERM_OF( objfunc, i )          TUPLE_SELECT_ITH( objfunc, 1+(i) )
#define ARG_OF( objgoal, i )              TUPLE_SELECT_ITH( objgoal, 2+(i) )
#define CONT_OF( objgoal )                TUPLE_SELECT_ITH( objgoal, 1 )
#define PRED_OF( objgoal )                TUPLE_SELECT_ITH( objgoal, 2 )
```

Figure 3.1: Sorts, indicators and selection operations for Prolog data types

efficient and robust mapping. "Efficient" means that the representation fits the functionality of the domain. It is essentially a question of nature. "Robust" means that every representation must be easily discriminated. It is a question of sort.

Natures and sorts are paired in *indicators* which have type T_INDIC. Indicators are structures with a sort field and a nature field. They must be computed by operation MK_INDIC which results in an indicator value. Fields must be selected with the selection operations INDIC_SELECT_SORT and INDIC_SELECT_NAT. Figure 3.1 shows definitions of indicators for terms that can represent Prolog data types. Indicators INDIC_FUNC and INDIC_GOAL have the same nature, N_TUPLE, but are disambiguated by their sorts.

The mapping of application terms onto MALIv06 is also required to define operations similar to the selection operations at the application level. Selection operations are given the reference of an object and return a pointer to one of its component cells. Figure 3.1 shows MPPM selection operations. Operation TUPLE_SELECT_ITH selects a subterm of a tuple. It has a supplementary parameter that is the index of the subterm.

3.1.1.2 MPPM make commands

The identifiers of MALIv06 make commands begin with MK_ and end with an indication of the nature of the term to be created. Make commands have parameters according to the specifics of each nature. They all have a parameter of type T_SORT indicating the sort of the created term. Make commands for compound terms have a parameter of type T_CELL * indicating the place where to store the name of the created term.

For non-degenerate[2] compound terms, the make command has a variable parameter of type T_ROBJ for returning a reference of a new object that represents the created term. The new object contains unfilled cells. Its reference must be used to complete the creation of the term. Make commands for tuples have also a parameter indicating the size of the term to be created.

MPPM make commands have identifiers beginning with MAKE_. Their parameters are similar to those of MALIv06 make commands, with specialised attributes added.

One must design a representation for Prolog terms and goals. We choose to copy goals and represent goal-statements as MALIv06 terms. A call to a make command of the MPPM corresponds to each symbol of a clause body, so that the sequential execution of the whole set of command calls makes a copy of the clause body and connects it to the tail of the goal-statement. Figure 3.2 shows a clause and the sequence corresponding to its body. Identifiers nrevS and appendS stand for the C static structures representing the Prolog predicate constants. Array X is an array of references to objects, and array U is an array of cells containing names of MALIv06 terms that represent Prolog unknowns. Indexes in X and U are allocated by the Prolog to C translator. The NGS and XG are registers of type T_CELL and T_ROBJ.

Arguments are indented so that those with similar types are aligned. To produce such a sequence is not difficult: it corresponds to the prefix Polish form of the clause body. The command name and the argument in the first column indicates the category and value (if any) of the corresponding symbol in the clause body. The second column argument constructs the subterm relation. The argument in the third column specifies how terms with subterms are identified in the subterm relation. The resulting term is designated by NGS. It has one unfilled cell for storing the name of the tail of the goal-statement. The unfilled cell is pointed to by CONT_OF(XG).

When invoked, every MALIv06 make command creates the representation of a term and invents a direct name of the term. The sort of the term and a pointer to a cell where to store the name must be given as parameters. The nature is given via the command iden-

[2] Every compound term, except nullary.

```
nrev  [A|L]  RLA :-
        nrev  L  RL,
        append  RL  [A]  RLA.

   MAKE_GOAL(                   &nrevS,    &NGS,         &XG   );      %  nrev(
           MAKE_UNKN(           &U[1],     ARG_OF(XG,1)        );      %       L,
           MAKE_UNK1(           &U[2],     ARG_OF(XG,2)        );      %       RL ),
   MAKE_GOAL(                   &appendS,  CONT_OF(XG),  &XG   );      %  append(
           MAKE_UNKN(           &U[2],     ARG_OF(XG,1)        );      %       RL,
           MAKE_LIST(                      ARG_OF(XG,2), &X[1] );      %       [
                   MAKE_UNKN(   &U[3],     CAR_OF(X[1])        );      %        A
                   MAKE_NIL(               CDR_OF(X[1])        );      %       ],
           MAKE_UNKN(           &U[4],     ARG_OF(XG,3)        );      %       RLA )

/* NGS = (tu  SORT_GOAL  4  (at SORT_SYMB &nrevS)
             (tu  SORT_GOAL  5  (at SORT_SYMB &consS)  CONT_OF(XG)@...
                  RL@(mc0 SORT_UNK)  (c2  SORT_LIST  A  (c0 SORT_NIL))  RLA)
             L  RL)
*/
```

Figure 3.2: A clause, the translation of its body, and the produced term

```
typedef struct { char *ident; int arity;                 } symbT;
typedef struct { char *ident; int arity; void (*pred)(); } psymbT;

void MAKE_SYMB(       value,        where )
               symbT *value; T_CELL *where;
{       ST_ATOM( where, SORT_SYMB, (T_INFO)value);
/*      where = (at  SORT_SYMB  value) */
}

void MAKE_UNK1(        var,         where )
               T_CELL *var; T_CELL *where;
{       MK_MCONS0( SORT_UNK, where );
        ST_CELL( var, where );
/*      var = where = (mc0  SORT_UNK) */
}

void MAKE_UNKN(        var,         where )
               T_CELL *var; T_CELL *where;
{       ST_CELL( where, var );
/*      where = var                 */
}

void MAKE_FUNC(       functor,        where,        obj )
               symbT *functor; T_CELL *where; T_ROBJ *obj;
{       MK_TUPLE( SORT_FUNC, where, *obj, 1+(functor->arity) );
        MAKE_SYMB( functor, FUNCTOR_OF(*obj) );
/*      where = (tu  SORT_FUNC  1+functor->arity  (at SORT_SYMB functor)  Args...) */
}

void MAKE_GOAL(        pred,        where,        obj )
               psymbT *pred; T_CELL *where; T_ROBJ *obj;
{       MK_TUPLE( SORT_GOAL, where, *obj, 2+(pred->arity) );
        MAKE_SYMB( (symbT*)pred, PRED_OF(*obj) );
/*      where = (tu  SORT_GOAL  2+pred->arity  (at SORT_SYMB pred)  Cont  Args...) */
}
```

Figure 3.3: Sample make commands

tifier. Commands MK_TUPLE and MK_MCONS0, which create nullary compound terms and mutable nullary compound terms, follow this scheme. They are used in commands MAKE_FUNC and MAKE_UNK1 for creating a Prolog compound term and a Prolog unknown (see figure 3.3).

In a first approximation, an unknown is a pure being with no information associated, but it is subject to substitution. So, the solution is to map it onto a mutable nullary compound. When decorated by types, constraints, or frozen goals, unknowns must be mapped onto larger muterms.

Procedure MAKE_UNK1 requires further explanations. It is expected that the identifier of an unknown has several occurrences in a clause. The first time the identifier of an unknown is encountered (here for copy), its representation must be created and its name must be remembered for the other occurrences. So, the name is stored in a register which is selected in a pool. The association between an identifier and a register is local to a clause, and is chosen by the Prolog compiler.

When an identifier of an unknown is encountered again, it remains to fetch the name of its representation in the associated register, and to plug it in the place designated by the where argument (see command MAKE_UNKN in figure 3.3).

Make commands creating non-degenerate terms have a variable parameter for returning the reference of the object that represents the created term. For instance, command MK_CONS2 creates a binary compound term and returns such a reference. It is up to the user to fill in the empty fields of the object. Command MK_TUPLE creates a n-ary compound term. It has a supplementary parameter indicating the size of the term. Command MAKE_FUNC (see figure 3.3) shows how to create a Prolog functional term. It allocates an object referenced by *obj, fills in one cell using reference *obj and selection operation FUNCTOR_OF, and leaves (functor->arity) unfilled cells. Structures representing function (resp. predicate) constant have type symbT (resp. psymbT). Pointer functor should point to such a structure.

A command MAKE_LIST, for creating Prolog lists, can be defined similarly using command MK_CONS2.

In a way that is symmetrical to the make commands, one can design MPPM read commands. A complete set of read commands is displayed in the MALIv06 tutorial. Once the mapping and the make and read commands are devised it is a good idea to make a set of display procedures. It is chiefly useful for debugging purposes.

3.1.2 Prolog control

The control of Prolog is mapped onto MALIv06 by means of two continuations. A search continuation (also called *failure continuation* or search-stack) implements *or-control*. It is mapped onto the term-stack. A goal continuation (also called *success continuation* or goal-statement) implements *and-control*. It is mapped onto terms (more precisely, on tuples).

The translation scheme amounts to associating a function to every predicate. The function is executed when a goal with the proper predicate constant appears at the beginning of the success continuation. It produces new continuations according to the goal and the associated predicate. A *motor* iterates calls to these functions until both continuations are empty. The control of Prolog is represented by the continuations plus the control of C. MPPM control commands only affect the continuations. C control statements must be used to complete the control operations.

Several registers are required to hold the context of the MPPM. Figure 3.4 shows their declarations. The general idea is that, when entering the predicate specified by register CP for solving the selected goal, the goal arguments are in array A, the tail of the goal-statement is in register GS, the clause counter is in register CC, the current choice-point is in register PCP, and register WMC contains 0. Arrays U and X are used in the scope of each clause to store unknowns and terms. The association between textual occurrences (identifiers of unknown and term notations) and indexes in these arrays is chosen by the compiler. Registers NGS and XG also serve in the scope of each clause for copying the clause body. Registers XR and GCR serve in the protocols for managing choice-points and calling the garbage collector.

3.1.2.1 Or-control

The search-stack of Prolog is mapped onto the term-stack of MALIv06. Choice-points are mapped onto levels and the information contained in choice-points must be saved in the root term.

A level is created and pushed on the term-stack by two separate commands. Command MK_TOP_LEVEL creates a level, given a sort and a root term to be saved. The new level is known internally as the *ghost level*. Command DO_PUSH_TOP_LEVEL pushes the ghost level on the term-stack. It has no parameter and stores the name of the new top-level in the only public register of MALIv06, TOP_LEVEL. As soon as the ghost level is pushed on the stack, its root term is protected from further substitution. It is as if a copy of the term were pushed on the stack.

Figure 3.5 shows the creation and updating of choice-points. Command TRY is used before the first clause of a non-deterministic predicate is entered. Its purpose is to save the current goal-statement and the updated current clause number on the search-stack. It must be followed by the C code corresponding to the first clause.

There is only one root name in a level, whereas several registers must be saved (current clause number, goal arguments and tail of goal-statement). So, a goal is created with the arguments and the tail goal-statement, and a root is created with the goal and the clause number. The root is implemented as a binary compound. Figure 3.5 shows operations applying on roots.

A term is considered saved on the term-stack until its level is popped. Command DO_POP_TOP_LEVEL pops the top level from the term-stack, sets the ghost level to be the popped level, and cancels the muterm substitutions that are more recent than the popped level.

Command FAIL_TRY (figure 3.6) shows how to handle a failure in a clause that is not the last clause of

```
T_CELL     GCR,                                /* The root for garbage collection  */
           GS,                                 /* The tail of the goal-statement   */
           NGS,                                /* The new tail                     */
           PCP,                                /* The parent choice-point          */
           A[many_cells], U[many_cells];       /* Arguments and unknowns           */
T_ROBJ     XR,                                 /* The root of the top level        */
           XG, X[many_robjs];                  /* A goal and terms                 */
int        CC,                                 /* The current clause               */
           WMC;                                /* The write mode counter           */
psymbT    *CP;                                 /* The current predicate            */
```

Figure 3.4: The MPPM registers

```
#define SORT_ROOT                  MK_SORT( 1 )
#define INDIC_ROOT                 MK_INDIC( N_CONS2, SORT_ROOT )
#define SORT_CHPT                  MK_SORT( 2 )
#define INDIC_CHPT                 MK_INDIC( N_LEVEL, SORT_CHPT )
#define CLAUSE_OF( objroot )       CONS_SELECT_1ST( objroot )
#define GOAL_OF( objroot )         CONS_SELECT_2ND( objroot )

void TRY()
{ T_CELL root; T_ROBJ objgoal; int arg;
        MAKE_ROOT( &root, &XR );
        CC = 2;
        MAKE_INT( CC, CLAUSE_OF(XR) );
        MAKE_GOAL( CP, GOAL_OF(XR), &objgoal );
        for ( arg=1; arg<=(CP->arity); arg++ ) {
                ST_CELL( ARG_OF(objgoal,arg), &A[arg] );
        }
        ST_CELL( CONT_OF(objgoal), &GS );
        MK_TOP_LEVEL( SORT_CHPT, &root );
        DO_PUSH_TOP_LEVEL();
}

/* After TRY, register TOP_LEVEL designates

        (le  SORT_CHPT
            (c2  SORT_ROOT
                (at  SORT_INT  CC)
                (tu  SORT_GOAL  2+CP->arity
                     (at SORT_SYMB CP)  A[1]  ...  A[CP->arity]  GS)
             old-top-level))
*/

void RETRY()
{       CC++;
        MAKE_INT( CC, CLAUSE_OF(XR) );
        DO_PUSH_TOP_LEVEL();
}

void TRUST()
{ }
```

Figure 3.5: Choice-points creation and updating

```
void FAIL_TRY()
{ T_ROBJ dummy;
        DO_POP_TOP_LEVEL( dummy );
}

void FAIL_TRUST()
{ T_ROBJ objlevel, objgoal; int arg;
        DO_POP_TOP_LEVEL( objlevel );
        READ_ROOT( LEVEL_SELECT_ROOT(objlevel), &XR );
        READ_INT( &CC, CLAUSE_OF(XR) );
        READ_GOAL( GOAL_OF(XR), &objgoal );
        READ_SYMB( &(symbT*)CP, PRED_OF(objgoal) );
        for ( arg=1; arg<=(CP->arity); arg++ ) {
                ST_CELL( &A[arg], ARG_OF(objgoal,arg) );
        }
        ST_CELL( &GS, CONT_OF(objgoal) );
        ST_CELL( &PCP, &TOP_LEVEL );
}
```

Figure 3.6: Backtracking

a predicate. Since the popped level becomes the ghost level, it is ready to be pushed again. There is no need to reinstall registers because they are not supposed to have changed. This is often called *shallow backtracking.* Command FAIL_TRY can be followed by a return statement to the motor, or by a branching statement to the next clause.

Command FAIL_TRY and other or-control commands implement a sequential search for unifiable clause heads. A common improvement in Prolog implementation is to use goal terms to compute keys giving a direct access to selected subsets of clauses. This is called *clause indexing.*

Command RETRY (see figure 3.5) shows how a choice-point popped off the search-stack by FAIL_TRY or FAIL_TRUST (see figure 3.6) is updated and resaved. Command RETRY must be used before executing every clause that is neither the first nor the last in a predicate. It must be followed by the C code for the clause.

To push a level exactly as it has been popped would cause a loop. A choice-point differs from the previous one by its clause number only. So, the root of the level is updated instead of creating a new level. Command MAKE_INT is used to have a side-effect on the root term. It is a rare instance in which it is justifiable to modify a MALIv06 term. A common experience in Prolog is that choice-point creation is costly; it is the same in MALIv06. So, it is crucial to avoid choice-point creation. In the MPPM, choice-points are made of a level, a root, and a goal.

When the last clause of a predicate is entered (see command TRUST in figure 3.5), there is no need to create a choice-point. To delete the top choice-point amounts to forgetting it, because there is no explicit destruction of terms. Anyway, it is still advisable to define and use this empty command because it shares with commands TRY and RETRY the property of being the first command of a clause. In a more sophisticated design, or for tracing the program, it might be non-empty.

Command FAIL_TRUST (figure 3.6) shows the more complicated case of a failure in the last clause of a predicate. One says that the predicate itself fails. This causes a *deep backtracking,* and all registers must be reinstalled according to the value of the popped choice-point. As above, before entering a predicate, command FAIL_TRUST saves register TOP_LEVEL in register PCP. It must be followed by a return statement to the motor.

3.1.2.2 And-control

And-control aims at selecting the first goal of the top goal-statement, for executing the function associated to its predicate constant, and so on. For efficiency reasons, the first goal (the head of the goal-statement) is not represented as the goals in the remainder of the goal-statement. A pointer to its predicate descriptor is stored in register CP and its arguments are stored in array A. The other goals (the tail) are represented in a term whose name is stored in register GS.

One of the purposes of the function associated to a predicate constant is to construct a new goal-statement. It is usually a slight variant of the old one, where the selected goal is replaced by the copy of a clause body. In the case of deep backtracking, the new goal-statement is an old saved goal-statement.

Copies of clause bodies are created by MPPM make commands. Their names are supposed to be stored in register NGS, while references to objects representing copies of goals are stored in register XG. There is only one register XG because goals are copied one after the other.

If there is only one goal in a clause body, the new goal replaces the former first goal and register GS need not be updated. It is enough to initialise registers CC and CP. Command JUMP_GOAL (see figure 3.7) implements this behaviour. The arguments are supposed to be properly installed using Prolog make commands and unification commands. So, the control of leftmost calls

```
void JUMP_GOAL(        pred )
               psymbT *pred;
{       CP = pred;
        CC = 1;
        CALL_GC();
        ST_CELL( &PCP, &TOP_LEVEL );
}

void LINK_AND_JUMP_GOAL(        pred )
                        psymbT *pred;
{       ST_CELL( CONT_OF(XG), &GS );
        ST_CELL( &GS, &NGS );
        JUMP_GOAL( pred );
}

void CONTINUE()
{ T_ROBJ objgoal; int arg;
        CC = 1;
        READ_GOAL( &GS, &objgoal );
        READ_SYMB( &(symbT*)CP, PRED_OF(objgoal) );
        for ( arg=1; arg <= (CP->arity); arg++ ) {
                ST_CELL( &A[arg], ARG_OF(objgoal,arg) );
        }
        ST_CELL( &GS, CONT_OF(objgoal) );
        CALL_GC();
        ST_CELL( &PCP, &TOP_LEVEL );
}
```

Figure 3.7: Management of the goal-statement

```
capsule :-
        query,
        success.
capsule :-
        exit.

void motor()
{       MAKE_NIL( &GS );
        CP = &capsuleS;
        CC = 1;
        WMC = 0;
        for (;;) (CP->pred)();
}
```

Figure 3.8: The capsule predicate and the Prolog motor

```
GS =   out@(tu SORT_GOAL 3 (at SORT_SYMB &outS)
              O@ga@(tu SORT_FUNC 2 (at SORT_SYMB &gS) (at SORT_SYMB &aS))
   success@(tu SORT_GOAL 2 (at SORT_SYMB &successS) (c0 SORT_NIL)))
A[1] = capsule2@(le SORT_CHPT
                  (c2 SORT_ROOT (at SORT_INT 2)
                     (tu SORT_GOAL 2 (at SORT_SYMB &capsuleS) (c0 SORT_NIL))
                _)
PCP =  in2@(le SORT_CHPT
              (c2 SORT_ROOT (at SORT_INT 2)
                 (tu SORT_GOAL 3 (at SORT_SYMB &inS) I@(mc0 SORT_UNK)
                 (tu SORT_GOAL 4 (at SORT_SYMB &transS) I O@(mc0 SORT_UNK)
                 (tu SORT_GOAL 3 (at SORT_SYMB &cutS) capsule2
              out@(tu SORT_GOAL 3 (at SORT_SYMB &outS) O@(mc0 SORT_UNK) success)))))
            capsule2)
CC = 1      WMC = 0      CP = &cutS
```

Figure 3.9: A control state

```
GS = success     A[1] = ga      PCP =  capsule2      CC = 1       WMC = 0       CP = &outS
```

Figure 3.10: Next control state

does not consume any terms.

After a clause body is created, and if it has more than one goal, register GS must be updated for inserting the new goals into the goal-statement. The reference of the rightmost goal is in XG. It has one cell left unfilled for storing its continuation. It is used to hook the tail of the goal-statement. Command LINK_AND_JUMP_GOAL (see figure 3.7) shows how the copy of the clause body is appended before the goal-statement. Commands JUMP_GOAL and LINK_AND_JUMP_GOAL can be followed by a return statement to the motor, or by a branching statement to the beginning of the procedure body.

When a clause body is empty —i.e. the clause is an assertion—, things are different because the next goal must be fetched in the tail of the goal-statement held in register GS (see command CONTINUE in figure 3.7). The first goal is read, and loaded in the registers. Register PCP is installed. Command CONTINUE must be followed by a return statement to the motor.

Throughout the control section, MALIv06's terms are read without checking for direct names (e.g. above, or READ_ROOT and READ_GOAL in command FAIL_TRUST). In fact, the designer of the MPPM knows that terms representing *and-control* never incur term replacement. This is not a general law. It may become false in a more sophisticated Prolog machine. Levels, which represent or-control, are subject to term assignment.

The reader may have noted that MPPM control commands never test if the search-stack or the goal-statement is empty. It is easy to obtain that they are never empty. Figure 3.8 shows a predicate that encapsulates the execution of every query. It must be translated in the same way as other predicates. So doing, the goal-statement always contains at least the goal success, and there is always at least a choice-point for the second clause of predicate capsule.

Predicate success always fails. It can print unknowns of the query and prompt the Prolog user for alternative solutions. Predicate exit stops the search. It can execute a postlude, or do an escape to a suitable environment. The actual query must be compiled as a clause of predicate query.

The capsule predicate can be much more sophisticated. In λProlog for instance, implication goals may be used to set environmental parameters of the execution.

The Prolog system must first execute a prelude, and then call the motor. To start the motor, a goal-statement is created, and the goal capsule is installed in the registers. The goal-statement must be initialised in GS, though it is never read, because no uncomplete term should either be saved on the term-stack or passed to the garbage collector. It is given the dummy value nil. The first step in the loop initialises the two continuations, and then the actual query is executed. It is the quintessence of an interpreter, but has almost nothing to do.

Figures 3.9 and 3.10 sum-up the control section. Given program

```
in (f a).
in (f b).

trans (f X) (g X).

out X :-
            ...
query :-
            in I,
            trans I O,
            /*1*/ !,
    out 0.
```

figure 3.9 shows the control state as it is when label /*1*/ is reached. Label 0 occurs in GS and PCP accompanied with different terms. Terms in GS differ by a substitution from terms with same labels in PCP, but they share the same representation anyway. Note that the (implicit) argument of goal ! is instantiated to a substack (labelled capsule2) of the search-stack. After goal ! is executed, the state is as shown by figure 3.10.

3.1.3 Prolog unification

The Prolog unification problem is to decide whether a *goal term* (selected in a goal-statement) and a *head term* (coming from the head of a clause selected in the program) unify, and, if it is the case, to apply the solution substitution to the goal-statement. Head terms are known as soon as a clause exists, and they are the same (up to a renaming of unknowns) for every instance of a unification problem with that clause. Goal terms are different in every problem instance.

Unification (either Prolog's of λProlog's) does not exist as such in MALIv06. It happens that, from a very operational point of view, unification is a mixture of reading, making, and substituting. Identifiers of unification commands begin with UNIF_.

Unification commands return a boolean value (in C, an integer). The value is true (different from 0) if the head and goal terms are compatible, and false (0) if not. The goal term is given explicitly as an argument, while the head is given implicitly as the unification command itself. Every unification command is specialised for one kind of Prolog term. Unification commands are intended to be aggregated in a conjunctive expression relating all the components of the head atom. According to the specification of C, the conjunction is evaluated left-to-right and it is exited as soon as an atomic expression evaluates to false.

Figure 3.11 shows a clause and the sequence corresponding to its head. Names of goal arguments are supposed to be stored in array A. Again, command arguments are indented to show their different roles. The unification sequence can be improved allocating unknowns X, Y and Z in array A instead of array U, but this is another story. The command identifiers indicate the kind of the head terms. The first column parameter (according to the indented layout) represents the binding of unknowns identified in the head terms. The second and third column parameters describe the subterm relation in the goal terms.

As it is explained above, unification commands are planned to be executed according to the head of a clause. For a given kind of head term, a unification command implements the suitable specialisation of the first-order unification algorithm. It shows the known part of a unification problem. The hidden part is the goal term. So, a unification command is specialised for a head term, but must allow for any goal term.

If the goal term is as instantiated as the head term, unification reads and compares. If the goal term is more instantiated than the head term, unification binds head unknowns to goal terms. These two behaviours form the *read mode*. If the goal term is less instantiated than the head term, unification must bind goal unknowns to head terms. However, goal unknowns are represented as MALIv06 terms, whereas head terms are represented by unification commands. So, one must construct a MALIv06 representation for the head terms. This is the *write mode*.

Fortunately, the two modes involve isomorphic data-flow. So, the same command with the same parameters can operate in the two modes.

Unification starts in read mode and switches to write mode as soon as it binds a goal unknown. The latency of the write mode is related to term structure. Unification switches back to read mode when the creation of a head term is completed. The MPPM controls mode switching via a *write mode counter* (WMC). It contains the number of unfilled cells in the created term.

Command UNIF_LIST (see figure 3.12) is the unification command for a Prolog list. Every time UNIF_LIST is executed in write mode, it creates a list constructor instead of reading one, it fills in one cell and produces two new unfilled cells. This is transposed in the incrementing of WMC.

It is essential that read and make commands are symmetrical so that they share the same parameter passing. Note that the program code for write mode is used when the command is entered in write mode and when it is entered in read mode but switches the write mode on. That is why the write mode code is not in the "else" branch of the "if" statement.

A successful unification may produce a term substitution, which is implemented with MALIv06 as a muterm substitution. Command SU_MUT substitutes a term, which must be specified later, to the muterm that is specified by its first argument. It returns in its second argument the reference of the object representing the muterm. The value field of the muterm must then be selected with the help of operation MUT_SELECT_VALUE to fill in the name of the binding value. The muterm is substituted though its binding value is not yet created.

After the creation of a muterm, its value field does not count as an unfilled cell. It is the substitution (or assignment) command that makes it an unfilled cell. When entering write mode, the unique unfilled cell is the value field of the muterm. That is why register WMC is initialised to 1.

Figure 3.8 shows that the motor initialises the write mode counter. Though commands TRY, RETRY and TRUST are always executed before unification, they need not reinitialise WMC because it is always 0 when unification succeeds or fails. Indeed,

1. unification commands cannot fail in write mode (i.e. when WMC is not 0), and
2. a properly generated unification sequence cannot end in write mode[3].

So, register WMC is always 0 when unification is exited.

[3]It is always a good idea to design a *debug mode* for executing the commands of the MPPM or any other intermediate machine. In debug mode, all sorts of checking can be performed: e.g. preconditions of the commands and invariants of the execution scheme. It helps debugging the implementation of the MPPM, and, more importantly, it helps debugging the code generation program.

```
append  [E|X]  Y  [E|Z] :-
        append  X  Y  Z .

        ( UNIF_LIST(                     &A[1],           &X[1] )         %  [
        &&      UNIF_UNK1(    &U[4],     CAR_OF( X[1] )         )         %   E|
        &&      UNIF_UNK1(    &U[1],     CDR_OF( X[1] )         )         %   X]
        && UNIF_UNK1(        &U[2],     &A[2]                  )         %  Y
        && UNIF_LIST(                    &A[3],           &X[1] )         %  [
        &&      UNIF_UNKN(    &U[4],     CAR_OF( X[1] )         )         %   E|
        &&      UNIF_UNK1(    &U[3],     CDR_OF( X[1] )         ) )       %   Z]
```

Figure 3.11: A clause and the translation of its head

```
int /* bool */ UNIF_LIST(          where,          obj )
                         T_CELL *where; T_ROBJ *obj;
{       if ( WMC == 0 ) {                                   /* read mode. */
                RD_DIRECT( where );
                switch ( where->indic ) {
                case INDIC_LIST :
                        READ_LIST( where, obj );
                        return 1;
                case INDIC_UNK :
                        { T_ROBJ objunk;
                                SU_MUT( where, objunk );
                                where = MUT_SELECT_VALUE( objunk );
                                WMC = 1;
                        }                                   /* enter write mode. */
                        break;
                default : return 0;
                }
        }                                                   /* write mode. */
        MAKE_LIST( where, obj );
        WMC += 2-1;
        return 1;
}

int /* bool */ UNIF_UNK1(          var,          where )
                         T_CELL *var; T_CELL *where;
{       if ( WMC == 0 ) {                                   /* read mode. */
                ST_CELL( var, where );
        }else {                                             /* write mode. */
                MAKE_UNK1( var, where );
                WMC += 0-1;
        }
        return 1;
}

int /* bool */ UNIF_UNKN(          var,          where )
                         T_CELL *var; T_CELL *where;
{       if ( WMC == 0 ) {                                   /* read mode. */
                return UNIFY( where, var );
        }else {                                             /* write mode. */
                ST_CELL( where, var );
                WMC += 0-1;
                return 1;
        }
}
```

Figure 3.12: Sample unification commands

Unification commands for other kinds of terms follow the same pattern. Though atoms do not involve a real term creation, their unification command is not a much simpler instance of the pattern because it must cope for the case in which the write mode is switched on in a previous command. For instance, command `UNIF_INT` never produces any unfilled cell and always consumes one; so, it decrements register `WMC`. When write mode is entered in the command, it does not last after the command invocation. The value of an atom must be cast properly before being compared. Otherwise, C might use the wrong instance of operator "==". Comparing floats is not the same as comparing pointers or integers.

Unification of head unknowns is again required to distinguish among first occurrences of identifiers of unknowns and others. For its first occurrence (see command `UNIF_UNK1` in figure 3.12), a head unknown cannot have been substituted. So, unification amounts to the assignment of the name of the goal term to a register in read mode, and to a creation of an unknown in write mode. In read mode, the unknown only serves as a coreference and is never created. The name pointed to by parameter `where` need not be made direct, because its indicator is not tested and it is not passed to commands requiring direct names.

For other occurrences (see command `UNIF_UNKN` in figure 3.12), neither the goal term nor the head term are known. Write mode amounts to the assignment of the head term to an unfilled cell of the goal term. In read mode, the complete unification procedure must be used.

3.2 Representation of terms in λProlog

To design a representation for terms in the context of λProlog is a new problem because the requirements of logic programming (Prolog technology), of the 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 unknowns and substitutions. It also requires that substitutions be reversible[4] because the search for a proof is done by a depth-first traversal of a search-tree.

The technology of the 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 requirement, long head-normalisation should be reversible too.

Proofs of hereditary Harrop formulas are required to represent universally quantified variables and to check the correction of signatures for a sound implementation of deduction rule $\forall_R$. Hereditary Harrop formulas also require the handling of implied clauses but it has little to do with the representation of terms.

[4] Strictly speaking, it is the operation that maps a substitution on the representation of some term which is reversible, not the substitution itself.
It the same for every other occurrence of reversible.

3.2.1 Typing

One of the differences between Standard Prolog and λProlog is that the terms of λProlog must be typed for λ-unification to be well defined.

3.2.1.1 Well-typed programs

The problem of what the typing of λProlog should be is not completely solved. We tend to try extrapolating from Mycroft/O'Keefe type system [56] (see also Typed Prolog [41]) to λProlog. The two basic ideas are that every clause should be well-typed in an ML-like fashion

> *Types of different occurrences of a constant are independent instances of its type scheme,*

and that the typing of predicate should obey the *definitional genericity principle*

> *Types of body occurrences of a predicate constant are independent instances of its type scheme, whereas types of head occurrences are only renaming of the type scheme.*

For the first idea, checking and inferring types are both decidable, as in ML [55], but for the second idea, only type checking is decidable. With the definitional genericity principle, type inference leads to a non-uniform semi-unification problem which has been shown to be undecidable by Kfoury, Tiuryn and Urzyczyn [39][5].

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. A sound and easy modular analysis of programs also requires definitional genericity. We want to be able to type-check a module using the type schemes of the modules it imports but not the modules themselves.

Example 3.2.1
Assume predicate constant append *is declared as*

```
type append
        A -> B -> C -> o.
```

in some module. The standard definition of predicate append *(see the "running example" in the introduction) violates the definitional genericity principle because occurrences of constant* append *in the heads have types that are strict instances of the type scheme. Using this type scheme is not enough for preventing an incorrect usage such as* (append [] 2 X).

Example 3.2.2
Assume predicate constant append *is declared as*

```
type append
        (list A) -> (list A) -> (list A) -> o.
```

and that clause

```
append "Lambda" "Prolog" "LambdaProlog".
```

[5] In our implementation, types of constants (predicative or not) are only checked, and types of unknowns, universal variables and λ-variables are inferred.

is added to the standard definition of predicate append. *It violates the definitional genericity principle because the type of constant* append *in the new clause*[6],

```
(list int) -> (list int) -> (list int) -> o
```

is a strict instance of the type scheme.

An expected outcome of a type theory for a programming language is a *semantic soundness* result that states that "well-typed programs cannot go wrong". In the context of λProlog, "going wrong" means "trying to solve ill-typed unification problems". The interest of such a result is to give formal grounds for not representing types at run-time. However, in λProlog, it is necessary to represent types at run-time for unification, and not all the conditions are met for having this kind of semantic soundness result.

3.2.1.2 Well-typed programs may "go wrong"

Type variables stand for arbitrary types. As for essentially existential variable, a type instance is never chosen arbitrarily. It is represented by a *type unknown* which gets a value through *type unification.* In $\lambda_\rightarrow$ with type variables, type unification is merely first-order unification.

The type declaration languages of ML or Typed Prolog share a common restriction that ensures the *type preserving property* [27]:

> *Every type variable in a type scheme should appear in the result type (the type to the right of the rightmost* ->*).*

Example 3.2.3
The declarations for constants cons *and* nil *(see section 1.1.1) can be written in ML as*

```
datatype 'a list = nil | cons of 'a * 'a list
```

In ML, type variables ("'a" in example 3.2.3) in the right side of "=" must occur in the left side. This ensures that well-typed programs cannot go wrong. This restriction makes it always possible to derive the type of subterms from the type of a term. λProlog lacks this restriction. This makes it possible that a well-typed program may "go wrong" if no dynamic type-checking is done.

In λProlog, this restriction could be imposed by saying that every type variable in a type declaration must occur in the result type. As for the interest of dropping the restriction, it may be a way for handling *dynamic types* [2] and we show in the following that it costs nothing for term constructors that obey the restriction.

Example 3.2.4
The following program goes wrong if no dynamic checking is done.

```
kind dummy
        type.
type eq
        A -> A -> o.
type forget
        A -> dummy.
query :-
        eq  (forget 1)  (forget x\x).
```

It is perfectly well-typed, but it goes wrong because procedure SIMPL eventually produces unification problem < 1, x\x >*, which is ill-typed.*

All this means that types (or, at least, parts of them) must be kept at run-time so that type unifiability can be checked before term unifiability. Remember that the λ-unification problem is only defined for equally typed terms. In fact, for solving this problem it is only necessary to represent the types which are instances of the type variables that do not appear in the result type of a declaration. We call these types the *forgotten types.*

Example 3.2.5
Constant forget *is implemented as if it were declared*[7]

```
type $forget
        'PI' A\(A -> dummy).
```

So, term (forget 1) *is implemented as* ($forget int 1)*, and term* (forget x\x) *is implemented as* ($forget (A->A) x\x) *for some* A.

If the unification procedure always checks type unifiability before trying to unify the subterms of a term, then type conflicts are always caught before an ill-typed unification problem is searched.

Note that, unlike Typed Prolog, there is no special syntax for declaring predicate constants. They are only distinguishable by their result type, o. So, 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 predicate obeys the definitional genericity principle, unification of these forgotten types always succeeds; type unification is only required for conveying types along the computation. In fact, if the type preserving restriction were imposed on functional constants only, well-typed programs could not go wrong [56, 27, 41].

To sum up, forgotten types of both function and predicate constants are represented at run-time. Type preserving constants (such as [] and '.') are implemented with no extra costs.

Note that when several instances of a forgotten type can be proven to be the same, then only one needs to be represented.

Example 3.2.6
In the program of example 3.2.1, it can be proven that the instances of type variable A *for both occurrences of*

[6] In λProlog as in Standard Prolog, quoted strings denote lists of integers that can be interpreted as character codes.

[7] This is not plain λProlog syntax. It is a wild extension towards more polymorphic types: here second-order types. Symbol 'PI' stands for the product type quantifier Π. Here, and in other occurrences, a $ sign prefixes internally defined symbols.

constant append *in the second clause are the same. So, they share the same representation. Note also that the internal constant implementing* append *is as if declared by*

```
type $append
        'PI' A\
        (   (list A) ->
            (list A) ->
            (list A) -> o
        ).
```

3.2.1.3 Projection is controlled by types

Unknowns must carry their types

As can be seen in figure 1.2, the projection rule specifies after the type of the flexible term which of its arguments may be selected for producing a substitution. This means that the type of all unknowns must be available at run-time. It can be either reconstructed or represented. We choose to have it represented.

It is still an open problem for us to see if an unknown can be compiled in a kind of "projector" according to its type. The projector of an unknown would select directly the eligible arguments for projection. Not every unknown is subject to this kind of compilation. Many of them are introduced at run-time by the imitation and projection operations of MATCH. They seem to be out of the scope of a compile-time analysis, though they may obey some pattern, anyway.

Types of constants must be computable

Procedure MATCH introduces new unknowns (the H_k's in figure 1.2) that must be typed properly. They all have types $\nu_1 \rightarrow \ldots \rightarrow \nu_p \rightarrow \rho$ where the ν_i's are the types of the arguments of the flexible head, and ρ depends on the rule.

In the case of projection (see notation in the condition controlling projection: 1.2), the ρ of every new unknown H_j is τ_j.

In the case of imitation, the ρ of every new unknown H_j can be inferred from the type of the rigid head. It is the type of the corresponding argument t_j. It remains to be able to infer the types of all rigid heads.

However, it is not necessary to have the rigid head carry its whole type. There are three kinds of rigid heads: λ-variables, function constants, and universal variables (they are introduced for solving universally quantified goals).

1. λ-variables cannot be imitated.
2. Every universal variable has its whole type attached to it (see section 3.2.3.2).
3. We observe that the type scheme of a constant, plus the forgotten types attached to it, plus the result type give enough information for reconstructing the type of the constant. During unification, the result type can be found in the type of the flexible head.

A type reconstruction function is generated at compile-time from every type scheme declaration. Note that the result type cannot be reconstructed from the forgotten types because it may contain instances of non-forgotten type variables (type variables occurring in the result type of the type scheme). So, the result type is passed to the type reconstruction function as a parameter.

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 $\sigma = \Pi\overline{\phi_m}.(\sigma' \; \overline{\phi_m})$[8], whose result type is ρ[9], we associate the type reconstruction function $\lambda\overline{\phi_m}\tau.(\mathcal{U}(\rho,\tau)(\sigma' \; \overline{\phi_m}))$. 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.

Example 3.2.7
Let the following declaration

```
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 $\lambda a\tau.\mathcal{U}($`(t B)`$,\tau)($`a -> B -> (t B)`$)$

With these declarations, unification problem `<(F 1),(f 1 [1])>` *is actually represented as*

```
        < (F:(int -> (t (list int))) 1)
        , ($f int 1 [1])
        > .
```

Imitation yields substitution

```
        [x\($f int (H1 x) (H2 x)) / F] .
```

which introduces new unknowns H1 *and* H2*. 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* [(list int) / B]*, 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)) *Then, unknowns* H1 *and* H2 *are given types* int -> int *and* int -> (list int)*.*

3.2.1.4 Some terms are dynamically η-expandable

Type variables add polymorphism to simple type theory. However, they cause the introduction of type unknowns, which do not mix friendly with normal form representations. Indeed, substituting a type unknown can change the arity of terms. Long head-normalisation of even rigid-terms is no more definitive, and dynamic η-expansion may be necessary.

[8] As suggested above, forgotten types are explicitly quantified.

[9] By definition, it cannot depend on forgotten types.

Example 3.2.8
The term $\lambda x.x = \mathrm{id}_V$ has type $V \rightarrow V$, where V is a type unknown, and it is in long head-normal form. If $\gamma \rightarrow \gamma$ is substituted to V, then the term must be η-expanded to $\lambda xy.(x\ y) = \mathrm{id}_{\gamma\rightarrow\gamma}$, whose type is $(\gamma \rightarrow \gamma) \rightarrow \gamma \rightarrow \gamma$.

Another lack of restriction in the language of type declarations prevents η-expanding every term at compile-time. It is possible to have a type variable as result type.

Example 3.2.9
Declaration

```
type identity
        A -> A.
```

is legal. However, an occurrence of constant `identity` *cannot always be η-expanded at compile-time because* `(identity 1)` *has type* `int` *and* `(identity cons)` *has type* `(t -> (list t) -> (list t))` *for some type* `t`.

When an unknown with unknown result type is used for projection, its arity (the number of arrows) is frozen for ensuring that no other projection is possible. This case is treated similarly by Nadathur [57] and Nipkow [60]. This is not really satisfactory and deserves further studies. A better solution (from a logical point of view) is to suspend unification.

3.2.1.5 The types that are actually represented

Only types of unknowns and universal constant, and forgotten types of constants are represented at run time. Types of constants are translated into type reconstruction functions.

A declaration of a type constructor is translated into the declaration and initialisation of a C structure for storing its external representation and its arity. The type of this C structure is given in figure 3.13.

Example 3.2.10
The C structure corresponding to type constructor `list` *(declared in example 1.1.10) is*

```
ksymbT  PM_k_list = { "list", 1 };
```

The representation of a type as a term of MALIv06 depends on what kind of type it is. Type constructions, arrows and type unknowns are respectively represented by tuples and atoms, binary compounds and nullary compounds. A type built with a type constructor of arity n (e.g. `list`) is represented as

```
(tu  SORT_TYPE_APPL  1+N
     (at SORT_TYPE_SYMB pointer)
     Type1  ...  TypeN)
```

A type built with a type constructor of arity 0 (e.g. `o`) is represented as

```
(at  SORT_TYPE_SYMB  pointer)
```

In both cases, `pointer` points to the C structure corresponding to the type constructor. A type built with an arrow is represented as

```
(c2 SORT_TYPE_ARROW Type1 Type2)
```

A type unknown is represented as

```
(mc0 SORT_TYPE_UNK)
```

Example 3.2.11
Type `(A -> A) -> (list int)` *is represented as*

```
(c2 SORT_TYPE_ARROW
    (c2 SORT_TYPE_ARROW
        a@(mc0 SORT_TYPE_UNK)
        a)
    (tu SORT_TYPE_APPL 2
        (at SORT_TYPE_SYMB &PM_k_list)
        (at SORT_TYPE_SYMB &PM_k_int)))
```

See section 2.1.1.2 for notation `a@(mc0 ...)`.

Note again (see example 3.2.6 in section 3.2.1.2), that when two or more of these types can be proven equal at compile-time, they need only be represented once. In recursive predicates for instance, argument types need only be represented once.

The types that are represented at run-time are involved in unification to different extents.

- Unification of types forgotten by predicate constants must be done before unifying the arguments and always succeeds because of definitional genericity.
- Unification of types forgotten by function constants must also be done before unifying the subtypes, but it may fail.
- Unification of types of unknowns need never be done. When an unknown enters a unification problem then it is certain that its type is the type of the other term of the problem. The type checking and unification have been done previously while unifying forgotten types of function constants.
- Unification of result types of unknowns with result types of type schemes is done in type reconstruction functions.

3.2.2 Representation of terms for Prolog

Two techniques are used for satisfying the traditional logic programming requirements: *structure-sharing* and *copy*. The former represents a term by the closure of a source term under an environment that binds its unknowns. The latter copies terms in order to have independent instances of them. Since our storage tool, MALI, makes it easier to do copy than to share structure, we choose copy.

With copy, the unit of usefulness[10] is the same as the unit of allocation; it is the term constructor. With structure-sharing, the unit of usefulness is smaller than the unit of allocation. An environment can be partially useless though it is not generally partially deallocatable because the useless part may be buried

[10] that to which predicate "is useful" applies.

```
typedef struct {                            /* Type constant: */
        char *ident;                        /* its external representation */
        int arity;                          /* its arity */
        } ksymbT;
```

Figure 3.13: Representation of type constructors

in the middle of the environment. The *environment trimming*[11] operation of the WAM [4] is an example of a partial de-allocation but it needs to know the goal-selection strategy. As soon as the goal-selection strategy is unpredictable, (constraints, delayed goals, ...), a lesser part of the environment is subject to trimming. This leads to memory leaks.

Note that either technique implies sharing of dynamic structures because multiple occurrences of an unknown are represented by multiple references to a single object in which substitutions are noted. Then the binding values are shared by all occurrences of the unknown. This sharing is much more fundamental than the sharing of structures.

3.2.3 Representation of terms for uniform proofs of hereditary Harrop formulas

Hereditary Harrop formulas introduce new operational features. Universal variables and λ-variables must be scoped, the lifetime of universal variables is bounded by some subproofs, clauses with still unknown terms in them might be added to the program, and these clauses also have a lifetime bounded by a subproof.

Many ways are possible for dealing with these features. Our preferred way makes implication share the term representation technology, and existential quantification a trivial issue. All the burden relies on universal quantification and on creation and substitution of regular unknowns.

3.2.3.1 The implementation of universal variables

The only source for universal variables is deduction rule $\forall_R$ which is used for solving universal quantifications in goals. The side-condition

> *c is a symbol that appears neither in P nor in G*

is the key of a correct implementation.

Our current implementation follows literally this deduction rule. Its side-condition can be rephrased as

> *c is a constant that neither appears in the P of the root, nor appears in a term introduced by a lower (closer to the root) instance of rules $\exists_R$ and $\forall_L$, nor is a constant introduced in a lower instance of rule $\forall_R$.*

Now, we can see that the lazy creation of terms for implementing rules $\exists_R$ and $\forall_L$ causes a problem: one cannot know what are the used constants[12]. A solution is to attach to unknowns (they stand for lazily constructed terms) the set of constants available at their creation time, the *allowed signature*.

Then, every time an unknown is substituted, the constants and unknowns of the substitution value are checked. Constants should be in the allowed signature of the unknown, and the allowed signature of every unknown of the value must be subsets of the allowed signature of the substituted unknown.

If it is taken literally, it means that the set of available constants should be known at every time. If one looks more closely, it appears that the sets of available constants of every sequent in a branch of a proof tree are totally ordered by inclusion, the smaller being at the root and the larger at the leaf. This means that rather than the sets, it is their sizes that are important and that must be attached to unknowns. So, universal quantification introduces universal variables that carry the size of the augmented signature.

It can be sketched in λProlog:

```
pi  G :-
        push_constant  NewSize,
        G  $universal_variable(NewSize),
        pop_constant.
```

Notation `$universal_variable(NewSize)` indicates that the universal variable is associated to the signature with size `NewSize`. In fact, the definition of `(pi G)` can be expanded in-line every time it is used in a goal position.

Now, every unknown must carry the representation of its allowed signature[13]. Then, every time an unknown is substituted, the universal variables and unknowns of the substitution value are checked against the signature size of the unknown. One checks that

- The size of the signature of every universal variable is lower or equal than the size of the signature of the unknown, and
- the same for the size of the signature of every unknown.

[11] If the goal-selection strategy is known, and if the occurrences of an unknown satisfy some conditions, it is possible to decide when its entry in an environment is certainly useless.

These unknowns are numbered such that they occupy an end of the environment when the entry becomes useless. Then the environment can be trimmed, simply shortening it.

[12] A bottom-up construction of proofs for hereditary Harrop formulas does not have this problem: terms have the right scope by construction. However, it has other problems [34].

[13] The size of the signature.

When the first condition is violated, substitution is impossible. When the second is violated, then the signature size of the conflicting unknown in the substitution value must be lowered until it fits the condition. We call this operation the *signature lifting.*

If an unknown with a conflicting signature is in fact in an argument of a flexible term then the scope-checking must be suspended because the problematical unknown may disappear as a side-effect of another substitution.

Note that when the terms are in L_λ, the signature checking is always decidable, because there can be no unknown in an argument of a flexible term.

Example 3.2.12
Substitution $[(U^1\ 1\ Y^2)/X^1]$ is problematical (the signature sizes are written as superscripts), but after substitution $[\lambda xy.(F^1\ x)/U^1]$ is applied, it is no more problematical. The term involved in the substitution is not in L_λ.

The signature checking can sometime be avoided. E.g. if the substituted unknown has the highest known signature then checking its binding value is useless. Unification commands must be augmented for dealing with signatures (see section 3.5.1). They are the places where this kind of situations can be encountered.

The representation of the current signature is considered in our execution model as a third continuation: the *signature continuation.* It has the same *search dynamism* as the success continuation. This means that it is saved on the search-stack and restored every time the success continuation is. To be saved on the search-stack a representation must be a term of MALI. In this case, an atom is enough for representing the size of the signature.

```
(at SORT_SIG sig)
```

3.2.3.2 Universal variables must carry their types

We have introduced a new kind of term: the universal variable. It behaves as a constant, and must be treated as such in the unification procedure (especially in the imitation rule). So, their types must be available in some way. But as opposed to constants, they are not declared and a type reconstruction function is not so easily produced. So, they carry their complete types at run-time.

They can be represented with MALIv06 as a binary compound term:

```
(c2 SORT_UNIV type (at SORT_SIG sig)).
```

A more compact representation uses unary compound terms and stores the allowed signature in the sort:

```
(c1 SORT_UNIV(sig) type).
```

The allowed signature is coded in the sort, and the type is the unique subterm.

3.2.3.3 The implementation of essentially existential variables

The two sources for essentially existential variables are rule $\exists_R$, which handles explicit existential quantifications in goals, and rule $\forall_L$, which handles implicit universal quantifications in clauses.

As we have seen in section 1.2, rules $\exists_R$ and $\forall_L$, in which a term is to be chosen, are implemented by introducing a new unknown.

This is our actual implementation of **sigma**.

```
type sigma
        (_ -> o) -> o.
sigma  G :-
        G  _ .
```

Again, this can be expanded in-line when (sigma G) is in a goal position, but a minimal carefulness is in order. One should refrain from in-lining a (sigma G) if its position as a goal results from the in-lining of a goal (pi G').

Example 3.2.13
A careless in-lining of goal

```
pi x\(sigma Y\
(   p  x  Y
))
```

results in

```
push_constant  NewSize,
p  $universal_variable(NewSize)  _ ,
pop_constant
```

which is a mistake. The scopes of variables x *and* Y *have been inversed because free variables (here,* _*) are implicitly quantified at the clause level.*

All the work is done in the resolution step which implements simultaneously deduction rules $\forall_L$ and $\supset_L$. Unknowns are created by commands like **UNIF_UNK1** and **MAKE_UNK1** (see section 3.1.3). The difference with the Standard Prolog case is that a nullary muterm is not enough for storing the representation of the type and of the allowed signature (see section 3.5).

3.2.4 Representation of terms for the λ-calculus

The main issue with meeting the requirements of the λ-calculus is β-reduction. The implementor has to answer a few technical questions.

3.2.4.1 Representation of β-reduction

The first question is

Does β-reduction alter the representation of terms, or not?

If yes, each time a procedure requires a long head-normal form of a term, it alters the representation of the term so that it becomes the representation of a λ-equivalent long head-normal term. If no, the representation of a term cannot be modified. A procedure that requires the long head-normal form of a term has to construct locally its own long head-normal representation; another procedure has to construct another one.

We prefer the first solution because it shares the β-reduction effort. To be consistent with the requirements of logic programming, the alterations of the representation by β-reduction must be reversible.

Note that what is merely an efficiency issue for a confluent system becomes a logical issue in a non-confluent one. If several non-confluent reductions are possible then all should be explored, and the reduced representation ought to be substituted to the non-reduced one for ensuring the coherence of the search.

3.2.4.2 β-reduction of representation

So, we choose that β-reduction alters the representation of terms. The second question is

> *How does β-reduction alter the representation?*

The representation to be computed must reflect the substitutions of a term to a variable in the leftmost term of a redex. One must avoid that the leftmost term of a redex be altered during its β-reduction. Indeed, it may have other occurrences that share the same representation. They should remain unaltered.

We know of two families of representations: environment based and graph based representations. In an environment based representation, the reduced term is represented by a closure of the leftmost term under an environment that binds its outermost variable. In a graph based representation, the reduced term is represented by a copy of the leftmost term in which the outermost variable is replaced by the other term.

We choose the graph based approach for the same reasons we have chosen the copy of terms: it is more convenient and efficient with MALIv06. With the graph based approach, reduction is called *graph-reduction*. Furthermore, graph-reduction is an efficient form of memory management. A new sharing is introduced by the graph based representation: the sharing of replacement values by the multiple occurrences of the variables.

MALIv06 memory management, especially muterm shunting, makes it sure that old versions of less reduced terms are not kept unduly in memory. This is done automatically, without the λProlog implementor having to take care of it. In Standard Prolog, muterm shunting has only the effect of shortening chains of unknowns and, so doing, eventually recovering the place occupied by the representation of unknowns (nullary muterms). In a Prolog with modifiable representation, muterm shunting shortens chains of non-reduced versions, and may recover the place occupied by the representation of intermediate versions (non-nullary muterms and their subterms). This is also the case for extensions to Prolog that introduce constraints, and dynamic proof strategy (e.g. the `freeze` predicate).

3.2.4.3 Representation of λ-variables

Third question is

> *How are the λ-variables represented?*

Two interesting possibilities are the nameless approach of De Bruijn [20], and a lexical representation which is almost isomorphic to the external syntax. The former is often described as unsuitable for a human being, but very suitable for a machine because it avoids the management of actual λ-variable names. The principles are to replace the multiple occurrences of the name of a λ-variable by numbers that specify the relative positions of the binding abstraction and the bound occurrences.

Example 3.2.14
De Bruijn's notation for term $\lambda x.(x\ \ \lambda y.(x\ \ y))$ *is* $\lambda.(0\ \lambda.(1\ 0))$.

De Bruijn's notation is not really compatible with a graph based representation because it forces to renumber the rightmost term —hence duplicate it or interpret it— for every occurrence of the replaced variable. The λ-variables free in the rightmost term are assigned different contexts for every occurrence of the replaced variable. So, the relative position of the free variables of the rightmost term and their binding abstractions may have to be updated.

It is often said that De Bruijn's notation is good for computers but terrible for humans. We think it is certainly good for mathematics, but can be terrible for humans *and* computers.

We choose a lexical representation of terms in which λ-variables have *pseudo-name*. Pseudo-names are references to the binding abstractions. But they are absolute references instead of relative references in the nameless scheme. Plainly speaking, pseudo-names have to do with memory allocation because they are implemented by references to data-structures.

3.2.4.4 Representation of unknowns

As we have seen in section 3.2.1.3, a type information must be added to unknowns. This type information is used to control projection during unification. They must also carry the representation of their allowed signatures; it can be coded in their sorts for immediate access.

Unknowns can be represented with MALIv06 as binary muterms:

```
(mc2 SORT_UNK type (at SORT_SIG sig)).
```

A more compact representation uses unary muterms and stores the allowed signature in the sort of the muterm:

```
(mc1 SORT_UNK(sig) type).
```

3.2.4.5 Representation of constants

Constants are represented according to their types (taken in the informal sense) as described in section 3.1.

For instance, λProlog integers can be represented with MALIv06 as

```
(at SORT_INT value)
```

More importantly, constants can be represented as

```
typedef struct {                                /* Function constant: */
        char *ident;                            /* its external representation */
        int arity;                              /* its arity */
        int type_arity;                         /* the number of its forgotten variables */
        void type_reconstruction();             /* its type reconstruction function */
        } symbT;

typedef struct {                                /* Predicate constant: */
        char *ident; int arity; int type_arity; void type_reconstruction();
        void (*pred)();                         /* the same, plus its predicate function */
        } psymbT;
```

Figure 3.14: Representation of constants

```
(at SORT_SYMB pointer)
```

where `pointer` points to a C structure describing the constant: arity, external representation for printing, predicate function if any, number of forgotten types, type reconstruction function, and even information for the debugging such as break-points, etc. Figure 3.14 shows the declaration of the C structure describing a function constant and a predicate constant.

Example 3.2.15
The C structures corresponding to constants `forget` *(declared in example 3.2.4),* `'.'`, `[]`, *and* `append` *(declared in example 1.1.10) are*

```
symbT  PM__forget =
        { "forget", 1, 1, PM_t_forget };
symbT PM__056 =                         /* '.' */
        { ".",   2, 0, PM_t_056 };
symbT PM__133135 =                      /* [] */
        { "[]",    0, 0, PM_t_133135 };
psymbT PM__append =
        { "append", 3, 1, PM_t_append,
          PM_c_append );
```

The prefixes (`PM__`, `PM_t_`, *and* `PM_c_`*) are added by the compiler to distinguish between the various "facets" of a symbol: the symbol itself, its type reconstruction function, and the function coding the relation when it is a predicate symbol.*

Then the constants are represented by

```
(at SORT_SYMB &PM__forget)
(at SORT_SYMB &PM__056)
(at SORT_SYMB &PM__133135)
(at SORT_SYMB &PM__append)
```

Note that to be type preserving or not is a property of constants, but that the extra cost for representing forgotten types is paid by applications that use the constants.

3.2.4.6 Representation of abstractions

The encoding of abstractions maps exactly the long head-normal form; all abstractions nested one under another are merged in one super-abstraction. So, $\lambda \overline{x_n}.t$, where t has an atomic type, is represented by a *(n+1)*-ary tuple of MALIv06. The first n arguments are the variables of the abstraction and the last is the body of the abstraction. So, what is actually represented as a unit is the binder of a term.

Each variable is represented by a place-holder whose only purpose is to be referenced. So, it can be as small as possible, but interesting duplication algorithms can be used if it is large enough to contain a name.

These interesting algorithms store the name of substitution values of variables in the place-holders, and then duplicate the term. This gives a direct access from a variable to its substitution value instead of associative search in a context. When duplication is over, the place-holders can be cleared.

λ-variables can be represented with MALIv06 as

```
(c1 SORT_VAR (c0 SORT_DUMMY)).
```

Because of type unknowns as result types, an abstraction can be dynamically η-expanded. So, the representation of an abstraction must also use a muterm. This enables modification of the abstraction, with an optimal memory management. With MALIv06, the term $\lambda \overline{v_n}.t$ is represented as

```
(mtu SORT_ABS N+1 v1 ... vN t).
```

3.2.4.7 Representation of applications

Applications also map exactly the long head-normal form. Like abstractions, they are encoded with n-ary tuples of MALIv06 representing the leftmost paths of the binary trees formed by the composition of abstraction. So, the internal representation of a term reflects the notation in which every unnecessary parenthesis are removed.

We call *potential redex* any application whose leftmost term is an unknown or a λ-variable. Non-potential redexes, also called *first-order applications*, have a fixed heading. So, they can be created in long head-normal form and always remain in long head-normal form.

In order to process them more efficiently, first-order applications are distinguished from potential redexes. Potential redexes may be β-reduced when their head, a λ-variable or an unknown, eventually gets substituted by an abstraction. So, the representation of a higher-order application uses a muterm to enable rewriting. Reducing a redex amounts to substituting its reduced form to the muterm. If the reduced form is again a potential redex, one must not forget to represent it by a new muterm. This is an efficient way to deal with normalisation while being consistent with depth-first search.

To sum up, the representation of a first-order application ($@ \overline{t_m}$), where constant @ has n forgotten types, is

```
(tu SORT_APP1 1+N+M
    @ type1 ... typeN
    t1 ... tM).
```

and the representation of a potential redex ($\overline{t_m}$) is

```
(mtu SORT_APP M t1 ... tM).
```

Note that there is no forgotten types in the potential redex. Indeed, forgotten types are associated to constants, hence, to first-order applications.

Some very frequent first-order applications are coded in a more compact way. For instance, lists nodes are coded like

```
(c2 SORT_LIST car cdr)
(c0 SORT_NIL) .
```

instead of

```
(tu SORT_APP1 3
    (at SORT_SYMB &PM__056)
    car
    cdr)
(at SORT_SYMB &PM__133135) .
```

The price of these compact representations is that specialised code must be written for dealing with them in any circumstances (unification —SIMPL and MATCH—, reduction, printing).

Example 3.2.16
We sum-up on representation of terms with a few examples. The list `[1,2]` *is represented as*

```
(c2 SORT_LIST (at SORT_INT 1)
(c2 SORT_LIST (at SORT_INT 2)
(c0 SORT_NIL))) .
```

The term `($forget int X)` *is represented as*

```
(tu SORT_APP1 3
    (at SORT_SYMB &PM__forget)
    int@(at SORT_SYMB_T &PM__int)
    x@(mc2 SORT_UNK int sig)) .
```

The term $\lambda nsz.(s\ (n\ s\ z))$ *is represented as*

```
succ@(mtu SORT_ABS 4
        n@(c1 SORT_VAR (c0 SORT_DUMMY))
        s@(c1 SORT_VAR (c0 SORT_DUMMY))
        z@(c1 SORT_VAR (c0 SORT_DUMMY))
        (mtu SORT_APP 2
            s
            (mtu SORT_APP 3 n s z))) .
```

3.2.5 Summary and comparison with other implementations

To sum up, we choose copy of terms for memory management reasons. We choose the alteration of the representation to share reduction effort, and also for memory management. We choose graph-reduction for memory management again. The λ-terms 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 reduced term, and then, of the reduction effort. "Reversibly" means that mutations (reductions) must be undone when backtracking. This is the result of plunging graph reduction in a Prolog context. Finally, a pseudo-name based representation is a natural consequence of the previous choices.

Every occurrence of a constant is represented with its forgotten types. The forgotten types of two occurrences must be unified before the arguments of the occurrences are unified. Every constant has a type reconstruction function associated to it. Type reconstruction functions take as parameters the forgotten types and result type of an occurrence of a constant, and return its complete type. Unknowns and universal variables are represented with their complete types. Types of unknown are used for controlling projection, and their result types are used for reconstructing types of constants. λ-variables do not carry any type information.

We know no publication on the implementation of either the original λProlog or eLP. So, we can only compare their behaviour with our implementation.

The implementation choices of a third λProlog implementation are described by Nadathur and Jayaraman [58, 36]. We can compare the choices but not the behaviour because we have no access to a running implementation. These choices are very different from ours, so that an experimental comparison seems necessary. The main differences are an environment based reduction scheme, and a nameless approach. Another of their ideas is to extend the WAM to execute λProlog. It is not relevant to the current discussion as long as memory management is not concerned. However, we think that to base a design on the WAM makes it tricky to obtain good memory management performances for λProlog terms and implication premises[14].

Kwon and Nadathur address the problem of having to represent types at run-time in a slightly different way [40]. They propose to represent occurrences of constants with *all* their type parameters, and to represent unknowns with their complete types. So, it seems that they considered the typing problem under the angle of the interpretation of polymorphic constants, whereas we considered it under the angle of the violation of the type preserving property. Note that the constructors of very important polymorphic data-structures (e.g. polymorphic lists) are type preserving and, then, incur no type representation at run-time in our scheme. Another difference is that their scheme allows for type reconstruction in every context, whereas ours allows for type reconstruction in unification only, or, at least in

[14] In fact, it might be as tricky as implementing MALI usefulness logic.

contexts in which the result type is available[15]. For the moment, unification is the only context in which type reconstruction is required. Finally, they rely on structure-sharing for representing types; what we call type reconstruction functions becoming type skeletons. This is only coherent with the starting point of their project: extending a structure-sharing version of the WAM[16].

Note also that differences are tempered as long as improvement are done to the original schemes. For instance, we use an environment based representation of terms for short-term storage inside unification and revert to the graph based representation for long-term representation outside unification. E.g. procedure SIMPL produces many short-lived terms that do not need a graph representation.

3.2.6 λ-quantification vs. ∀-quantification

It is interesting to note that λ-quantification and ∀-quantification can represent each other. However, we choose to have separate specialised implementations for both. It appears that they share lower-level routines.

This correspondence is merely folklore and we do not use it directly in our implementation. However, it can be used for justifying some implementation techniques (see procedure TRIV in section 3.5.6.2). The λProlog programmer must also keep this correspondence in mind because a lot of λProlog programming is about converting essentially universal quantification at the term level (λ-quantification) towards universal quantification at the goal level (∀-quantification), and *vice versa* (see for instance predicate `has_type` in example 1 or first version for predicate `list2flist` in section 1.5.2).

As the two quantifications are represented concurrently, it would be more precise to speak of "λ∀-unification" and "λ∀-equivalence".

3.2.6.1 λ-quantification for representing ∀-quantification

λ-unification can implement the side condition of rule $\forall_R$ through skolemisation. The idea is to replace rule $\forall_R$ by the variant shown in figure 3.15. It introduces a new kind of goals quantified by a λ-abstraction. The other deduction rules must be revisited to deal with such goals on the basis of procedure SIMPL. It means that deduction rules must propagate λ-abstractions from formulas to subformulas, just like procedure SIMPL propagates λ-abstractions from terms to subterms.

In example 3.2.17, the application of procedure SIMPL is anticipated in the program: every logical variable is applied to the λ-variable that stands for a universal variable. This is careless with respect to types, but we hope it makes the example clearer. This make skolemisation visible in the program.

[15] It is always possible to reconstruct types of terms from types of goals, but we hope to never have to consider this.

[16] Note that the WAM is not at all committed to structure-sharing [75].

Example 3.2.17
The following program illustrates rule $\forall_R^\lambda$.

```
type eq
        A -> A -> o.
eq  X  X.
query :-
        pi x\
        (   eq  x  Y
        ),
        print  Y.
```

The query fails because goal `(eq c Y)` *should be proven for a constant* `c` *that must be different from every other constant, including the value of unknown* `Y`*, whereas clause* `(eq X X)` *forces them to be equal. It is easy to transform the goal and program so that λ-unification implements the side-condition.*

Suppose that the program[17] *is transformed into*

```
x\(eq (X x) (X x)).
query :-
        x\(eq  x  Y),
        print  Y.
```

Solving the new query is required to solve unification problem

`< x\(eq x Y), x\(eq (X x) (X x)) >` ,

which simplifies to

`{< x\x, x\(X x) >, < x\Y, x\(X x) >}` ,

and η-reduces to

`{< x\x, X >, < x\Y, X >}` ,

The first pair is trivially solved by substitution `[x\x / X]`*, yielding the derived problem*

`< x\Y , x\x >`

which is detected unsolvable by MATCH because unknown `Y` *is not allowed to capture λ-variable* `x`.

To understand why the transformation works, remember the intuition given in section 1.3.2 for flexible terms. In the situation of example 3.2.17, term `(X x)` is an unknown term in which λ-variable `x` can occur, and `Y` is an unknown term in which λ-variable `x` cannot occur. Miller gives the formal basis of this kind of manipulation under the name of "unification under mixed-prefix" [51].

In a real-life implementation, the program is not actually modified. The set of universal variables, such as `x` above, is in the search context and unknown `X` is interpreted as `(X x)` according to the context. As noted by Nadathur and Jayaraman, the set of universal variables has a stack-like life and can be denoted by a unique number, its size.

An early stage of our implementation of λProlog followed this scheme. We found it less efficient than the technique of the signature size. We think it is because this specialised usage of λ-unification deserves a more specialised procedure. Even if the signature-checking is a costly operation, it is tailored for one purpose. This makes it less inefficient than λ-unification.

[17] "Goal", "query", and "program" are now used in a broad sense, because they have no more type o.

$$\frac{\lambda x.P_x \vdash \lambda x.G}{P \vdash \forall x.G}\ \forall_R^\lambda$$

P_x is the result of substituting in P every essentially existential variable v by $(v\ x)$. If needed, variable x can be renamed with α-conversion.

Figure 3.15: The rule $\forall_R^\lambda$

```
SIMPL$^\forall$: $(\Lambda \times \Lambda) \rightarrow (2^{(\Lambda\times\Lambda)} \cup$ Failure)
SIMPL$^\forall$($< t^1, t^2 >$) =
        assume $t^1 = \lambda\overline{u_n}.(@_1\ \overline{e^1_{p_1}})$ and $t^2 = \lambda\overline{v_n}.(@_2\ \overline{e^2_{p_2}})$
        in if $n \neq 0$
           then SIMPL$^\forall$($< (t^1\ \overline{c_n}), (t^2\ \overline{c_n}) >$)
           else if $@_1 \in \mathcal{U}$
                then $\{< t^1, t^2 >\}$
                else if $@_2 \in \mathcal{U}$
                     then $\{< t^2, t^1 >\}$
                     else if $@_1 \neq @_2$
                          then Failure
                          else $\cup_{i\in[1\ p_1]}$ SIMPL$^\forall$($< e^1_i, e^2_i >$)
```

Every c_k is a new universal variable with the appropriate type.

Figure 3.16: Procedure SIMPL$^\forall$

3.2.6.2 $\forall$-quantification for representing λ-quantification

In procedure SIMPL, one can suppress the need for accumulating abstractions by applying these abstractions to new universal variables. Note that this is correct *only* when η-equivalence is assumed.

Example 3.2.18
If η-equivalence is not assumed, terms $t_1 = \lambda xy.(x\ y)$ and $t_2 = \lambda x.x$ are not λ-equivalent, but terms $(t_1\ u\ v)$ and $(t_2\ u\ v)$ are β-equivalent for every u and v.

Figure 3.16 shows the modified version of SIMPL. Note that "new universal variable" means among other things that the signature is augmented.

Example 3.2.19
The query of program

```
type eq
        A -> A -> o.
eq  X  X.
query :-
        eq  v\(U 1)  v\(g (v 1)).
```

fails because unknown `U` *has to capture λ-variable* `v` *for making the two terms equal. It is easy to see that projection gives nothing because* `< v\1, v\(g (v 1)) >` *is detected unsolvable by SIMPL. We concentrate on imitation. It substitutes* `w\(g (H w))` *to unknown* `U` *yielding a new unification problem,*

```
< v\(g (H 1)), v\(g (v 1)) >
```

which simplifies to

```
< v\(H 1), v\(v 1) >
```

which is clearly unsolvable because unknown `H` *cannot capture λ-variable* `v`.

It is easy to transform the unification problem so that $\forall$-quantification controls the capture of λ-variables. The unification problem is

```
< v\(U 1), v\(g (v 1)) > .
```

We apply the two terms to the same *new universal variable* `$`*, yielding the new problem*

```
< (U 1), (g ($ 1)) > .
```

New *means that* `$` *is in the scope of* `U`*. Again, projection can do nothing, and imitation substitutes* `w\(g (H w))` *to unknown* `U`*. The new unknown* `H` *is created in the scope of universal variable* `$`*, but since it is in the binding value of* `U` *it must migrate to the scope of* `U`*.*

The new unification problem is

```
< (g (H 1)), (g ($ 1)) > ,
```

which simplifies to `< (H 1), ($ 1) >` *and is unsolvable. Substitution* `[w\($ (G w)) / H]` *cannot be a solution because now that unknown* `H` *has migrated to the scope of* `U` *the universal variable* `$` *is no more in its available signature. So, unknown* `H` *cannot be substituted by a term containing universal variable* `$`.

So the side-condition of rule $\forall_R$ prevents capture of λ-variables. Again, we prefer a specialised procedure for a specialised purpose. We never tried this implementation, but its most visible inconvenient is that it costs a β-reduction for every abstraction that procedure SIMPL traverses. A direct implementation of procedure SIMPL avoids the β-reductions using a temporary environment based representation.

3.3 Implication

Expression $P \wedge D$ in rule $\supset_R$ (see figure 1.5) shows that the program is augmented in some way when an implication goal is executed. However, the implementation of this rule cannot use the Standard Prolog assert/retract technology (see section 1.4.3).

We call *static* the clauses that belong to the program since compile-time, and *dynamic* clauses that are added to the program when executing an implication goal. We call *dynamic* the predicates that can be extended by executing an implication goal, *static* the other predicates.

3.3.1 From a premise-stack to a closure-stack

In a first design of our implementation of λProlog, facts (or clauses) that are dynamic clauses (i.e. premises of implication) were pushed on a *premise-stack* when entering the proof of the conclusion. They were popped when returning from the proof. This can be expressed in Prolog as

```
D => G :-
        push_clause  D,
        G,
        pop_clause.
```

As for quantifiers pi and sigma, this can be expanded in-line. Dynamic clauses are searched for before static clauses. This is done by a special first clause that is added to every predicate. To avoid paying an excessive cost, dynamic predicates must be declared. So the search in the premise-stack and the special clause are paid only by predicates that deserve it.

The premise-stack is considered as an outgrowth of the goal-statement and undergoes the same operations with respect to the management of the non-determinism. Both are pushed and popped at the same time on the search stack. Since dynamic clauses are shared with the goal-statements, pushing involves no copying and amounts to consing. Recall that predicate assert must copy the clause to protect it from further substitutions or backtracking (i.e. to close the term by a new quantification).

In a latter design (somewhat inspired by Nadathur and Jayaraman's work [36]), dynamic clauses are compiled as (almost) ordinary clauses. At run-time, only their connection with the proof context is managed as we did in the first design. So, the premise-stack is replaced by a *closure-stack*. The only difference with static clauses is that care is taken for handling the environment of the dynamic clause (i.e. the context of the corresponding implication goal).

This is done by the means of supplementary parameters and of supplementary universal quantifiers.

1. A dynamic clause $\forall\overline{x}.(\Phi \subset \Psi)$, for predicate p, with free variables $\overline{y_m}$[18], is translated into the static clause $\forall\overline{y_m}\,\overline{x}.((p'\ \overline{y_m}\ \Phi) \subset \Psi)$, 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. The actual values of the free variables are added to the arguments of every goal that is solved using this clause. Then comes the actual head of the dynamic clause.
2. When the clause is added to the program[19], a closure is stored. It links constant p with constant p' in a context made of the current values $\overline{c_m}$ of variables $\overline{y_m}$. It has the form $< p, \lambda g.(p'\ \overline{c_m}\ g) >$.
3. When a goal $(p\ \overline{s})$ is executed, the closure-stack is searched for all the closures whose left-hand part is p. Their right-hand parts are successively applied to the goal, and the resulting goals called.

Example 3.3.1
In implication goal

```
        (   pi Z\
            (   p  X  Z  (f Y) :-
                     q  W
            )
        =>  G
        )
```

the dynamic clause is compiled as

```
type $this_p
        TX -> TY -> TW -> o -> o.
pi X\(pi Y\(pi W\(pi Z\
(   $this_p  X  Y  W  (p X Z (f Y)) :-
        q  W
)))).
```

or in the implicitly quantified notation

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

where $this_p *is associated to this clause of the predicate.*

Predicate $this_p *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* $this_p *for making the compilation easier. Another solution is to pass its parameters separately.*

So, instead of storing an entire clause in a premise-stack, a pair

```
        < p, g\($this_p X Y W g) >
```

is stored in the closure-stack. The pair associates the predicate constant p *to the new constant* $this_p *in the environment* X Y W.

A closure is conceptually an abstraction. When selected, it is applied to the current goal for building a new goal (of constant $this_p in example 3.3.1) which is executed using the single clause for it. Note that since the abstraction is very specialised (e.g. there is always one occurrence of λ-variable g, and it is always in the same position) it need not be actually implemented as an ordinary abstraction.

[18] Note that free variables may be either universal (e.g. pi x\(p x => q)) or existential (e.g. (p X => q)).

[19] I.e. when the corresponding implication goal is executed.

The closure-stack (or the premise-stack) is implemented so that a hash-table may give almost direct access to relevant clauses (contexts) given a predicate constant. However, no clause indexing is implemented (neither in the all system).

The closure-stack is considered in our execution model as a fourth continuation: the *program continuation*. It has the same search dynamism as the success continuation or the signature continuation.

3.3.2 Implication and definitional genericity

Implication is in contradiction with definitional genericity because it may add to the program clauses whose head symbol has a type that is a strict instance of the type scheme. Since we wish to stick as much as possible to definitional genericity, there is a problem which is still open.

Example 3.3.2
Assume predicate constant append *is declared as*

```
type append
        (list A) -> (list A) -> (list A) -> o.
```

nothing prevents adding a clause with a more instantiated head symbol in the following way:

```
(    append  "Lambda"
             "Prolog"
             "LambdaProlog"
=>   ...
)
```

3.3.3 Comparison with other implementations

The implementation of implication by Nadathur and Jayaraman is explained at length in their ICLP'91 paper [36]. They compile the clauses that are in an implication goal as ordinary static clauses, and use a new kind of choice-point (remember that their scheme is WAM-based) for representing the instances of them that are currently in the program. The connection to unknowns that were introduced in the program is made using the environment in which the implication goal was executed.

Their scheme is made slightly more complicated than ours by the problem of adding, removing, and re-adding clauses due to backtracking. This is transparent using the muterms of MALIv06.

3.4 Reduction procedure

The normalisation of λ-terms is essential for the unification procedure. It takes the place of the dereferencing function *deref* in Standard Prolog implementations [3]. Function *deref* goes through bindings of unknowns. It is very similar to command RD_DIRECT and can be implemented by it in a MALI based implementation of Standard Prolog. To transform non-direct names into equivalent direct names is a trivial form of normalisation. This is not enough for λProlog.

Normalisation is required to β-reduce redexes and η-expand the unknowns and λ-variables.

We show in the following that reduction in λProlog is basically lazy, then we present a powerful optimisation which finds its root in the new properties of unknowns and substitutions presented in section 1.1.3. Finally, we show how the representation of terms can be folded.

3.4.1 Lazy outermost reduction

In fact, except for making displayed terms more readable, it is never necessary to completely reduce a term. β-reduction is done only when required, i.e. before unification, to allow comparison of terms. Even when unification is required to do β-reduction, it is not required to achieve a complete normalisation. What it is required to do is to put the terms to be unified in long head-normal form, so as to exhibit the headings of the terms. It follows that β-reduction is done with an outermost strategy. Note that outermost reduction is not necessary to have a converging normalisation of simply typed λ-terms. In this context, it is only used for its connection with lazyness.

Example 3.4.1
A common form of function-lists (see the "running example") follows the pattern `z\[e1|(L z)]` *rather than the normal form pattern* `z\[e1,...,eN|z]`*:*

```
z\[1|(z\[2|(z\[3|z] z)] z)]
```

instead of the normal form `z\[1,2,3|z]`*. The two versions of predicate* `list2flist` *produce lists which have the first form. In both forms, the first element of the list and the end of the list are accessible in constant time.*

A third pattern is `z\(L [eN|z])`*:*

```
z\(z\(z\(z\z [1|z]) [2|z]) [3|z]) .
```

Predicate `fnrev` *produces a list with this form. With this representation, the end of the list is accessible in constant time, but an access to the first element is required to reduce all the redexes. It is linear because of the detection of combinators (see in section 3.4.2). Note that after only one access to its first element, a list has its structure normalised (its elements may still be non-normal). So, further accesses are immediate.*

3.4.2 Detection of combinators

Because the left member of a β-redex may be shared by some other term, β-reduction cannot apply its substitution *in situ*. It is required to duplicating the left member.

Duplicating subterms whose only free λ-variables are also free in the redex is useless. We want to detect as much as possible of this circumstance.

It would be too cumbersome to record for every term all the λ-variables that occur free in it and in some including redex. It is much easier, if less precise, to record the terms having no free λ-variable at all. These terms are usually called closed terms or combinators. So, all closed subterms of the left member of a redex are shared by reduction.

A lot of terms are combinators. Every instance of a combinator is itself a combinator because λ-unification forbids λ-variables capture. So, it is effective to record which source terms are combinators. Every binding value is a combinator by definition of λ-unification. So, binding values are tagged as combinators as soon as they are created. Then, combinators detection incurs no dynamic checking cost.

Our experience is that the recognition of *all* source combinators, and the tagging as combinators of *all* binding values is crucial. The mere exception causes a visible slow down. Note that a lot of terms are recognised to be combinators only because we have distinguished λ-variables from unknowns in the definition of λ-terms in section 1.1.1.2. If this distinction was dropped, non-groundness and non-closedness would be mixed and no "interesting"[20] term could be recognised as a combinator.

The effect is three-fold. Less time is spent in β-reduction. Less memory is consumed, hence less time is spent garbage collecting. More sharing is achieved, hence unification and β-reduction costs are better factorised.

Constant terms, unknowns, and universal variables, which are always combinators, need no tagging. λ-variables, which are never combinators, need no tagging either. So, there remain applications and abstractions which are tagged in their sorts. With MALIv06,

```
(mtu SORT_ABS(COMB/NO_COMB) N+1
     v1 ... vN t)
(mtu SORT_APP1(COMB/NO_COMB) 1+N+M
     @ type1 ... typeN
     t1 ... tM).
(mtu SORT_APP(COMB/NO_COMB) M
     t1 ... tM).
```

This improvement is fundamental and changes the complexity of useful λProlog predicates [15] (e.g. predicate fnrev for reversing a function-list; see definition in section 1.5.1 and complexity comparisons in figure 4.7). It is not committed to our architecture; it only has to do with reduction.

Note again that combinatorness, which is about occurrences of λ-variables, cannot be completely decided for flexible terms. A closed flexible term remains closed anyhow, but a non-closed flexible term may become closed at run-time. We do not try to recognise this kind of combinators at run-time.

Example 3.4.2
Term $(U\ \ x)$, where U is an unknown and x is a λ-variable, is not a combinator, but after substitution $[\lambda x.1/U]$ is applied it is a combinator.

3.4.3 Folding representations

Both β-reduction and η-reduction may cause folding of representation.

Example 3.4.3
Term $t_1 = (\lambda x.(f\ x)\ 1)$ β-reduces to $(f\ 1)$. We assume the type int → int *for constant f.*

[20] In logic programming, interesting terms are not ground.

Figure 3.17 gives a graphical representation of the term before and after β-reduction. Allocation of new terms and representation sharing can be observed. Nodes are decorated by the terms they represent and optionally by labels (e.g. t1@ *$(f\ 1)$). Nodes with identical labels are the same node before and after the β-reduction.*

Note the potential redexes represented as muterms. Every muterm is graphically represented with an arc labeled by a circle. When the muterm is not substituted, the circle is empty and the arc points to its compound term part. When the muterm is substituted, the circle is filled and the arc points to its substitution value.

The terms are represented with MALIv06 as follows:

- $(\lambda x.(f\ x)\ 1)$ *: (this entry and the next one share the same label space)*

```
t1@
(mtu  SORT_APP(COMB)  2
      (mtu  SORT_ABS(COMB)  2
            x@(c1  SORT_VAR
                   (c0  SORT_DUMMY))
            fx@(tu  SORT_APP1(NO_COMB)  2
                    fff@
                    (at  SORT_SYMB  f)
                    x))
            one@(at  SORT_INT  1))
```

The outermost application is represented as a muterm because it has been a potential redex sometime. The innermost application is represented as a non-mutable term because it is not a potential redex. The COMB/NO COMB *tags are set by an analysis prior to the execution.*

- $(f\ 1)$ *:*

```
t1@(tu  SORT_APP1(COMB)  2  fff  one)
```

Note that the reduced representation takes the place of the old one and that subterms labelled by fff *and* one *are shared.*

In this example, they are both atoms so that sharing is not a great matter. However, any actual argument or combinators of the left-hand of the redex with a non-atomic representation would be shared as well.

The structure of the reduced term t1 *derives from term* fx*, but the tag* COMB *is inherited from the tag of the non-reduced* t1.

Example 3.4.4
Term $t_2 = \lambda x.x$ η-expands to $\lambda yz.(y\ z)$. Figure 3.18 gives the graphical representation.

The terms are represented with MALIv06 as follows:

- $\lambda x.x$ *: (this entry and the next one share the same label space)*

```
t2@
(mtu  SORT_ABS(COMB)  2
      x@(c1  SORT_VAR  (c0  SORT_DUMMY))
      x)
```

The abstraction is represented by a muterm because $\lambda x.x$ is a potential η-redex.

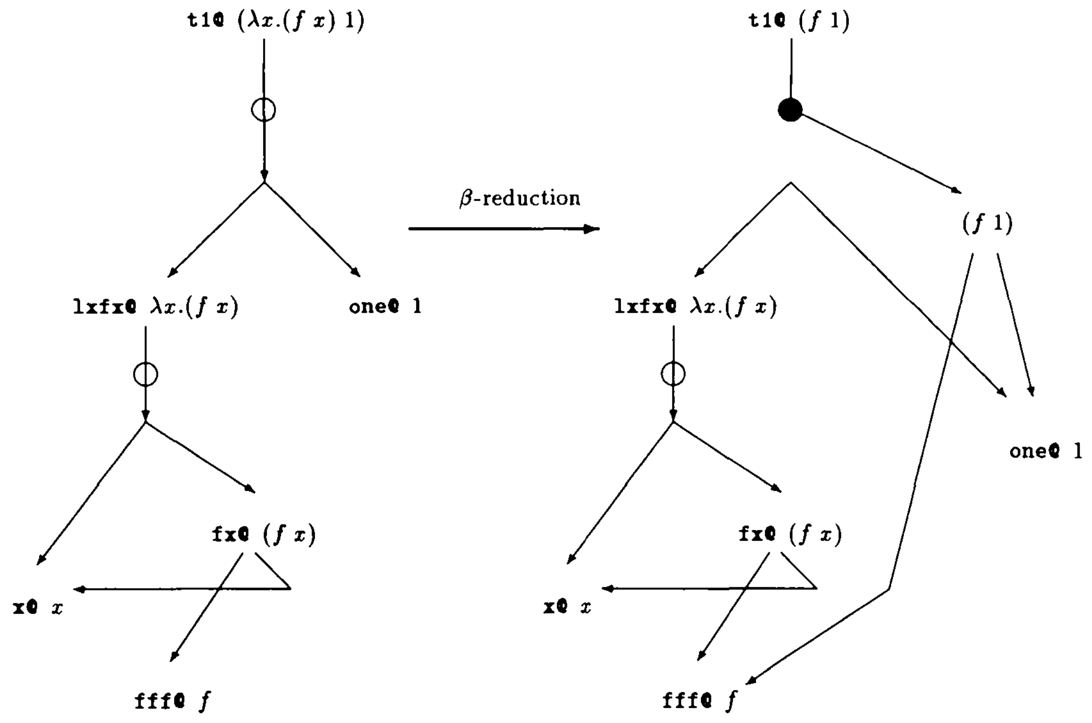


Figure 3.17: β-reduction of $(\lambda x.(f\ x)\ 1)$ into $(f\ 1)$

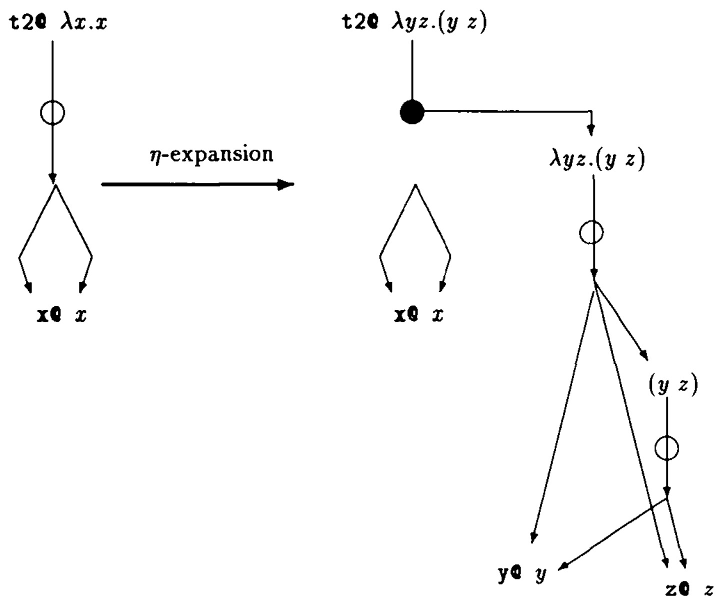


Figure 3.18: η-expansion of $\lambda x.x$ into $\lambda xy.(x\ y)$

- $\lambda yz.(y\ z)$:

```
t2@
(mtu  SORT_ABS(COMB)  3
      y@(c1  SORT_VAR  (c0  SORT_DUMMY))
      z@(c1  SORT_VAR  (c0  SORT_DUMMY))
      (mtu  SORT_APP(NO_COMB)  2  y  z))
```

Note that the η-expanded representation takes the place of the old one. As in example 3.4.3, the expanded representation inherits tag COMB *from the original term.*

With a memory management point of view, one must note that if a reduction is done definitively (no backtrack point over it), the representation of the redex $((\lambda x.(f\ x)\ 1)$ in example 3.4.3) may be collected according to the usefulness logic of MALI.

3.5 Unification

We present now the implementation of the unification procedure. It is made of a collection of unification commands, and of procedures UNIF (first-order unification), SIMPL, TRIV and MATCH. Finally we show that, like reduction, unification gives a chance for folding representation.

Everything starts from the unification commands that are generated for the clause heads. Then, as things get more complicated, procedure UNIF and the others are called. They are organised as a sequence of sieves that try to solve a given unification problem or pass it to the next sieve if it is not in their capability. So doing, we give the highest priority to simple specialised problems.

3.5.1 Compiled unification

As we have seen in the MPPM (see section 3.1.3), unification can be partially compiled when one of the terms of a unification problem is known. But, even with first-order unification, one has to revert sometimes to the general procedure (see command UNIF_UNKN). The terms of λProlog can be compiled in a similar way.

3.5.1.1 Read/write mode

In Standard Prolog, unification commands operate in read mode or write mode according to the relative instantiations of the goal term and the head term. Write mode is used for building the binding value of an unknown coming from the goal term. In λProlog, write mode is also used when read mode makes no sense because of the properties of the λ-unification problem. So, write mode is also used to build the representation of unification problems to be solved later.

Operating in write mode for creating a binding value is a trivial operation in the first-order case, but it is less trivial in the higher-order case because of the signature checking. In principle, unknowns should be created with the current allowed signature. However, they are created as parts of the binding value of an unknown that already have an allowed signature. The naive solution is to try creating the unknowns of the binding value with the allowed signature of the substituted unknown. It does not work because flexible application may make the signature lifting incorrect.

The problem is that the write mode behaviour is in fact a compilation of the TRIV procedure. It is always permitted in Standard Prolog, but we have seen that flexible applications make any decision on occurrences unsafe (e.g. signature checking). A rough solution is to compare the signature of the unknown to be substituted and the current signature. If they are equal, the unknown is substituted (TRIV is permitted), otherwise, a unification problem is created dynamically whose members are the unknown and the candidate binding value. In both cases, the write mode is entered, either for building a binding value, or for building a term of a unification problem. Figure 3.19 shows sample commands of the MPPM upgraded for dealing with allowed signature. Command ADD_UNIF_PAIR adds the new unification problem to the collection of not yet solved problems.

Because the general λ-unification is too complex, unification commands for flexible terms or abstractions never operate in read mode. In this case also, the commands operate in write mode for building a unification problem. The method for creating a unification problem is the same as in command UNIF_LIST. Since these unification commands always operate in write mode, they are mere variants of make commands.

Note also the management of the combinator tag in figure 3.19. Because of previous remark, unification commands always operate on head terms that are combinators. We cannot assume that goal terms are tagged COMB.

3.5.1.2 Type unification

As we have seen in section 3.2.1.5, some types are represented at run-time. Type commands must be designed for either creating or unifying them.

Types represented at run-time belong to two categories: forgotten types and types of unknowns.

Forgotten types behave much like extra parameters to term constructors. The only difference is that their unification order is specified: they must be unified before the regular parameters. Since they are first-order terms, the technique described in the MPPM applies directly.

The type of an unknown is only part of the representation of the unknown. It should be created if and only if the unknown is created. So, there is no read mode for types of unknowns, and make commands for types are used in the unification sequences. Figure 3.20 shows the compilation of the head of the recursive clause of append. Commands MAKE_TYPE_FUNC, MAKE_TYPE_UNK1 and MAKE_TYPE_UNKN follow the patterns of commands MAKE_FUNC, MAKE_UNK1 and MAKE_UNKN of the MPPM. Note that, like for unknows, first and next occurrences of type unknowns must be distinguished.

Variables X, Y and Z have the same type but three different sequences handle the three occurrences of the type. It does not mean that the same type is created three times; in mode (append + + -), only the type of variable Z is created. In mode (append + + +), no type

```
#define SORT_LIST(comb)             MK_SORT((comb?2001:2002))
#define INDIC_LIST(comb)            MK_INDIC(N_CONS2,SORT_LIST(comb))
#define SIG_OF_UNK(objunk)          CONS_SELECT_2ND(objunk)
#define TYPE_OF_UNK(objunk)         CONS_SELECT_1ST(objunk)
#define COMB                        1
#define NO_COMB                     !COMB

int       AS;                      /* Allowed signature    */

void MAKE_LIST(                 comb,         where,        obj )
               int /* bool */ comb; T_CELL *where; T_ROBJ *obj;
{       MK_CONS2( SORT_LIST(comb), where, *obj );
/*      where = (c2 SORT_LIST(comb) ... ...) */
}

void MAKE_UNK1(        var,          where,         obj )
               T_CELL *var; T_CELL *where; T_ROBJ *obj;
{       MK_MCONS2( SORT_UNK, where, *obj );
        ST_CELL( var, where );
        MAKE_SIG( AS, SIG_OF_UNK(*obj) );
/*      var = where = (mc2 SORT_UNK ... (at SORT_SIG sig)) */
}

int /* bool */ UNIF_LIST(       where,        obj )
                         T_CELL *where; T_ROBJ *obj;
{       if ( WMC == 0 ) {                               /* read mode. */
                RD_DIRECT( where );
                switch ( where->indic ) {
                case INDIC_LIST(COMB) : case INDIC_LIST(NO_COMB) :
                        READ_LIST( where, obj );
                        return 1;
                case INDIC_UNK :
                        { T_ROBJ objunk, objpair; int sigunk;
                                READ_UNK( where, &objunk );
                                GET_SIG( &sigunk, SIG_OF_UNK(objunk) ) ;
                                if ( sigunk < AS ) {
                                        ADD_UNIF_PAIR( &objpair );
                                        ST_CELL( PAIR_SELECT_1ST(objpair), where );
                                        where = PAIR_SELECT_2ND(objpair);

                                }else {
                                        SU_MUT( where, objunk );
                                        where = MUT_SELECT_VALUE(objunk);
                                }
                                WMC = 1;
                        }                                /* enter write mode. */
                        break;
                default : return 0;
                }
        }                                                /* write mode. */
        MAKE_LIST( COMB, where, obj );
        WMC += 2-1;
        return 1;
}
```

Figure 3.19: Management of allowed signature

```
type append
       (list A) -> (list a) -> (list A) -> o.

append [E|X] Y [E|Z] :-
       append X Y Z .

       ( UNIF_LIST(                          &A[1],                  &X[1] )
       &&      UNIF_UNK1(         &U[4],     CAR_OF( X[1] ),         &X[2] )
       &&      ( WMC == 0
               || ( MAKE_TYPE_FUNC(  &listS,   TYPE_OF_UNK(X[2]),      &X[3] ),
                    MAKE_TYPE_UNK1(  &U[5],    ARG_OF_TYPE(X[3],1)           ),
                    true ))
       &&      UNIF_UNK1(         &U[1],     CDR_OF( X[1] ),         &X[2] )
       &&      ( WMC == 0
               || ( MAKE_TYPE_FUNC(  &listS,   TYPE_OF_UNK(X[2]),      &X[3] ),
                    MAKE_TYPE_UNKN(  &U[5],    ARG_OF_TYPE(X[3],1)           ),
                    true ))
       && UNIF_UNK1(              &U[2],     &A[2],                  &X[2] )
       && UNIF_LIST(                         &A[3],                  &X[1] )
       &&      UNIF_UNKN(         &U[4],     CAR_OF( X[1] )                )
       &&      UNIF_UNK1(         &U[3],     CDR_OF( X[1] ),         &X[2] )
       &&      ( WMC == 0
               || ( MAKE_TYPE_FUNC(  &listS,   TYPE_OF_UNK(X[2]),      &X[3] ),
                    MAKE_TYPE_UNKN(  &U[5],    ARG_OF_TYPE(X[3],1)           ),
                    true )) )
```

Figure 3.20: Management of types of unknowns

is created.

3.5.2 UNIF

Procedure UNIF performs first-order unification as much as possible. It is almost redundant because SIMPL and MATCH can unify first-order terms. Note however that first-order unification can deal with flexible-flexible pairs whereas MATCH cannot. Procedure UNIF is used for executing the first-order part of λProlog programs as efficiently as in implementations of Standard Prolog. The main difference with Standard Prolog is that long head-normalisation takes the place of function *deref* (see 3.4), and that types must be accounted for anyway. First-order unification is used as often as possible. It is the only part of unification that is compiled, i.e. specialised instances of it are computed.

When a λ-unification subproblem pops up during first-order unification, the pair of terms to be unified is created and stored, and first-order unification continues with other unification subproblems. When a term of a λ-unification problem comes from the head of a clause, it is created by a sequence of unification commands always operating in write mode. If unification ends with a success, the stored pairs are given to SIMPL.

This is a general scheme for dealing with the unification of new domains. We can imagine different unifications procedures which could be called by UNIF according to the domain of terms.

3.5.3 SIMPL

SIMPL, which is the deterministic part of λ-unification is not compiled in the current state of our implementation of λProlog. However, it could be compiled (almost) as well as UNIF.

The implementation of SIMPL mirrors its formal description. The list of pairs to be given to MATCH is constructed like a λProlog list. Procedure SIMPL does not call directly MATCH because MATCH is implemented as a λProlog predicate. When needed, SIMPL prepends a call to MATCH in the goal-statement. Note that when it is straightforward that MATCH is deterministic (e.g. a flexible-rigid pair with no argument in the flexible term) SIMPL computes the substitution without the help of MATCH.

Unification of types is always done before unification of terms (see 3.2.1.5), so that SIMPL always see well-typed problems.

3.5.4 MATCH

MATCH takes as argument a list of flexible-rigid pairs and solves them one after another. For each pair, the imitation and projection procedures are non-deterministically called. They perform substitutions of the flexible heads and call SIMPL with the new pairs obtained.

Imitation and projection introduce new unknowns. For projection, the types of these unknowns can be deduced from the type of the substituted unknown. For imitation, their types are deduced from the type of the rigid head. If the rigid head is a constant, its type is reconstructed by the function which is generated for every constant declaration. If it is a universal variable, it carries its own type. It cannot be a λ-variable.

MATCH is written in λProlog whereas it is called from unification. This kind of predicate call amounts to insert a goal in front of the goal-statement that is created after the body of the clause.

Inserting MATCH goals prevents (or at least makes less easy) improvements of the execution control that are otherwise easy to implement. In the WAM as with the MPPM, the first goals of a clause are not compiled like the others. The first built-in's are executed in-line if they are deterministic. The first non-built-in is not created, but its arguments are stored into registers, ready for the next execution step. MATCH goals come across these improvements because, when inserted, they must be checked immediately, and they are non-deterministic.

So, even if the improvements are implemented (which is in fact the case), one must always check that no MATCH goal has been awakened. If a MATCH goal must be executed then the in-lined goals and the in-registers goal must revert to the regular representation.

3.5.5 Flexible-flexible pairs as constraints

Constraints are not difficult to implement, at least with MALI. It is only a special case of merging terms and control.

The flexible-flexible pairs are added like constraints to the two unknowns which are the heads of the terms. Note that every constraint must contain the representation of its allowed signature because it might be awakened in a different context[21]. The flexible-flexible pair is added to the attributes of the unknowns (their types and allowed signatures, plus other constraints if any). When one of the unknowns is eventually substituted, the constraints are processed.

This is similar to the ***attributed variable*** technique described by Le Huitouze [43].

3.5.6 TRIV

TRIV is a heuristic for solving more problems than the general procedure can handle. In many cases, a unifying substitution can be straightforwardly computed. Two difficulties arise: the need for an occurrence-check and the fact that an unknown may be hidden in an abstraction by η-expansion. A failure of TRIV does not imply a failure of unification. It only means that the general procedure must be applied.

3.5.6.1 The occurrence-check problem

The most trivial case is a unification problem with the form $< X, t >$. A solution is the substitution $[t/X]$ if X and t satisfy an occurrence-check. The notion of occurrence-check for λ-unification is not as simple as for first-order unification. An occurrence of X in t may be discarded by further reduction. Sufficient criteria for performing the trivial unifications are presented by Huet.

[21] More generally, a suspended computation must be represented with its relevant program and signature continuations. Suspended unification needs only the signature continuation because it does not depend on the program.

Example 3.5.1
Let term t be $(U\ X\ Y)$, X could appear in t according to whether U gets substituted by $\lambda xy.x$ or $\lambda xy.y$.

As for Standard Prolog, the occurrence-check is not implemented. However, a special first-order equality predicate with occurrence-check is available for doing type-checking in λProlog. Occurrence-checking is really crucial for type-checking.

3.5.6.2 Dealing with η-equivalence

η-expansion may cause an unknown X to be replaced by $\lambda\overline{u}.(X\ \overline{u})$. An efficient TRIV procedure must recognise similar cases. When `fnrev` is called in mode `(fnrev + -)`, its second argument is an unknown but it is η-expanded. The procedure TRIV recognises it and does the substitution. The permutation version of TRIV [57] is not implemented in our system, but it can be.

Another feature of TRIV comes from the observation that the application of an unknown X with allowed signature S to a collection of universal variables $\{\overline{u}\}$, disjoint from S, is equivalent to an unknown with allowed signature $S \cup \{\overline{u}\}$. TRIV recognises cases of the form $< (X\ \overline{u_n}), t >$ and computes substitution $[(\lambda\overline{v_n}.t[v_1/u_1, \ldots, v_n/u_n])/X]$. In fact, the duality of λ-quantification and $\forall$-quantification shows that it is another case of recognising η-equivalence.

The latter situation is very frequent. It occurs every time there is a conversion between an essentially universal quantification at the term level and an essentially universal quantification at the goal level.

Example 3.5.2
In clauses

```
abstraction E :-
        pi x\
        (  variable  x
        => term  (E x)
        ).
```

and

```
list2flist L FL :-
        pi list\
        (  append  L  list  (FL list)
        ).
```

terms `(E x)` *and* `(FL list)` *are recognised as unknowns by procedure TRIV.*

Because term t may contain unknowns and because it is not certain that term $\lambda\overline{v_n}.t[v_1/u_1, \ldots, v_n/u_n]$ is really useful, the substitutions $[v_i/u_i]$ are not actually applied to t. They are coded as ***explicit substitutions*** [67, 1]. Note that long head-normalisation done before λ-unification has better perform immediate substitutions because the heading of the term is actually used by λ-unification. In this case of TRIV, one cannot be so sure that the normal form is useful. Since explicit substitutions must be reduced sometime, they are represented by muterms:

```
(mc3 SORT_SUBST u v t)
```

As we have seen in section 1.3.5, recognising L_λ patterns generalises a lot of trivial pairs. It subsums all the above examples. Recognising flexible L_λ patterns is not more complicated than recognising more specific TRIV patterns such as η-redexes. Recognising rigid L_λ patterns is much more complicated; it has the complexity of an occurrence check. Recognising rigid L_λ patterns matters less than recognising flexible L_λ patterns because as soon as one term of a unification problem is rigid then Huet's procedure applies and is deterministic if the terms are in L_λ. Recognising flexible L_λ patterns allows to apply elimination of flexible-flexible pair when Huet's procedure can only suspend the problem.

3.5.7 Folding representations

3.5.7.1 Folding unificands

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 to 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-equivalence substitution). Reversibility comes as a consequence of using muterms. Folding saves unification effort because identity of representation is easier to check than equality. It also saves memory, hence garbage collection time.

Note also that, when used systematically, folding of unificands is the basis of rational terms unification [18, 22, 73, 42] and of a fast unification procedure [31].

3.5.7.2 Folding normal forms

Terms in unification problems must be put in long head-normal form before being compared. After applying the substitutions invented by imitation or projection, the flexible term is 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 to the head of the flexible term, compute its new long head-normal form, and substitute it to the flexible term.

Remembering that unification also substitutes equal for equal, it appears that many more substitutions than the so-called "solution substitutions" are done. The supplementary substitutions contribute to saving unification and reduction time, and to saving memory.

Example 3.5.3
Unification problem $< t_1, t_2 >$, *where* $t_1 = \lambda x.t_3$, $t_3 = (U\ (x\ S_1))$, *and* $t_2 = \lambda x.(x\ S_2)$, *yields three substitutions after one run of MATCH:*

1. $[\lambda y.y/U]$ *(solution substitution),*
2. $[(x\ S_1)/t_3]$ *(for direct long head-normalisation of* t_1 *before passing it to SIMPL), and*
3. $[t_2/t_1]$ *(substituting equal for equal).*

Remember that unknowns, abstractions, and potential redexes are all represented by muterms. So, they are mutable.

3.6 Special static patterns

Compiling becomes really valuable when special source patterns exist for which the general execution scheme can be specialised. We sum up in this section the patterns our compiler currently recognises and the associated specialisations and savings.

3.6.1 Forgotten types

A simple example (see in section 3.2.1.2) shows that well-typed programs may go wrong if types are not dynamically checked. Moreover, computation of projection is based on types. So, types must be represented at run-time.

Representing types of unknowns solves the projection problem but does not help preventing programs from going wrong. More types need to be represented. The worst would be to have to represent the types of every term and subterm. The important pattern that improves the representation of type 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 done at compile-time indicates which types are identical and can share representation.

3.6.2 Combinators

β-reduction is required to duplicate left members of redexes. It is easy to see that combinators need not be duplicated and that 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.

3.6.3 First-order applications and constants

The general unification procedure of λProlog is Huet's algorithm. However, first-order applications (i.e. applications with a constant leftmost term) and first-order constants in head are recognised for using the first-order unification scheme. So, these patterns are compiled rather classically. The representation of first-order applications is chosen so as to be easily recognisable so that, at run-time, unification and β-reduction are improved.

```
a\b\(SUCC c\d\(SUCC e\f\(SUCC ZERO g\(e g) f) h\(c h) d) i\(a i) b)                 1
a\b\((a ((SUCC e\f\(SUCC ZERO g\(e g) f) h\(c\(i\(a i)c)h) b))))
a\b\((a (((h\(c\(i\(a i)c)h) (e\f\(SUCC ZERO g\(e g) f) c\(h\(c\(i\(a i)c)h)c) b))))))
a\b\((a (((c\(i\(a i)c) (e\f\(SUCC ZERO g\(e g) f) c\(h\(c\(i\(a i)c)h)c) b))))))     2
           ^^^^^^^^^^^^

a\b\((a (((i\(a i) (e\f\(SUCC ZERO g\(e g) f) c\(h\(c\(i\(a i)c)h)c) b))))))
           ^^^^^^^

a\b\((a (((a (e\f\(SUCC ZERO g\(e g) f) c\(h\(c\(i\(a i)c)h)c) b))))))
a\b\((a (((a ((SUCC ZERO g\(c\(h\(c\(i\(a i)c)h)c)g) b)))))))                       3
a\b\((a (((a (((g\(c\(h\(c\(i\(a i)c)h)c)g) (ZERO c\(g\(c\(h\(c\(i\(a i)c)h)c)g)c) b)))))))))
a\b\((a (((a (((c\(h\(c\(i\(a i)c)h)c) (ZERO c\(g\(c\(h\(c\(i\(a i)c)h)c)g)c) b)))))))))   4
                ^^^^^^^^^^^^^^^^^^^^^^

a\b\((a (((a (((h\(c\(i\(a i)c)h) (ZERO c\(g\(c\(h\(c\(i\(a i)c)h)c)g)c) b)))))))))
                ^^^^^^^^^^^^^^^^^

a\b\((a (((a (((c\(i\(a i)c) (ZERO c\(g\(c\(h\(c\(i\(a i)c)h)c)g)c) b)))))))))
                ^^^^^^^^^^^^

a\b\((a (((a (((i\(a i) (ZERO c\(g\(c\(h\(c\(i\(a i)c)h)c)g)c) b)))))))))
                ^^^^^^^

a\b\((a (((a (((a (ZERO c\(g\(c\(h\(c\(i\(a i)c)h)c)g)c) b)))))))))
a\b\((a (((a (((a b))))))))
```

Figure 3.21: The reduction of *(SUCC (SUCC (SUCC ZERO)))*

```
a\b\(SUCC c\d\(SUCC e\f\(SUCC ZERO g\(e g) f) h\(c h) d) i\(a i) b)                 1
a\b\((a ((SUCC e\f\(SUCC ZERO g\(e g) f) h\(c\(i\(a i)c)h) b))))
a\b\((a (((h\(c\(i\(a i)c)h) (e\f\(SUCC ZERO g\(e g) f) c\(h\(c\(i\(a i)c)h)c) b))))))
a\b\((a (((c\(i\(a i)c) (e\f\(SUCC ZERO g\(e g) f) c\(h\(c\(i\(a i)c)h)c) b))))))     2
a\b\((a (((i\(a i) (e\f\(SUCC ZERO g\(e g) f) c\(h\(c\(i\(a i)c)h)c) b))))))
a\b\((a (((a (e\f\(SUCC ZERO g\(e g) f) c\(h\(c\(i\(a i)c)h)c) b))))))
a\b\((a (((a ((SUCC ZERO g\(c\(h\(c\(i\(a i)c)h)c)g) b)))))))                       3
a\b\((a (((a (((g\(c\(h\(c\(i\(a i)c)h)c)g) (ZERO c\(g\(c\(h\(c\(i\(a i)c)h)c)g)c) b)))))))))
a\b\((a (((a (((c\(h\(c\(i\(a i)c)h)c) (ZERO c\(g\(c\(h\(c\(i\(a i)c)h)c)g)c) b)))))))))   4
a\b\((a (((a (((h\(c\(i\(a i)c)h) (ZERO c\(g\(c\(h\(c\(i\(a i)c)h)c)g)c) b)))))))))
a\b\((a (((a (((c\(i\(a i)c) (ZERO c\(g\(c\(h\(c\(i\(a i)c)h)c)g)c) b)))))))))
a\b\((a (((a (((i\(a i) (ZERO c\(g\(c\(h\(c\(i\(a i)c)h)c)g)c) b)))))))))
a\b\((a (((a (((a (ZERO c\(g\(c\(h\(c\(i\(a i)c)h)c)g)c) b)))))))))
a\b\((a (((a (((a b))))))))
```

Figure 3.22: The reduction of *(SUCC (SUCC (SUCC ZERO)))* with $\overline{\beta}$-reduction

```
a\b\(SUCC c\d\(SUCC e\f\(SUCC ZERO g\(e g) f) h\(c h) d) i\(a i) b)                 1
a\b\((a ((SUCC e\f\(SUCC ZERO g\(e g) f) i\(a i) b))))
a\b\((a (((i\(a i) (e\f\(SUCC ZERO g\(e g) f) i\(a i) b))))))                       2
a\b\((a (((a (e\f\(SUCC ZERO g\(e g) f) i\(a i) b))))))                             3
a\b\((a (((a ((SUCC ZERO i\(a i) b)))))))
a\b\((a (((a (((i\(a i) (ZERO i\(a i) b)))))))))
a\b\((a (((a (((a (ZERO i\(a i) b)))))))))                                          4
a\b\((a (((a (((a b))))))))
```

Figure 3.23: The reduction of *(SUCC (SUCC (SUCC ZERO)))* with $\overline{\beta}\eta$-reduction

3.6.4 Long head-normalisation of source terms

All terms of source clauses are put in head-normal form before generation. All terms, except logical variables, are η-expanded according to their types. This makes dynamic long head-normalisation less necessary. It also provides a macro-like feature which may improve the programming style. Long head-normalisation makes more applications to be first-order applications.

However, a careless use of η-expansion costs several orders of complexity in some cases. The problem is that the number of β-redexes and the size of the terms that must be duplicated during β-reduction are dramatically increased.

Example 3.6.1
If no care is taken with η-expanded terms, the time complexity of goal

```
SUCC = n\s\z\(s (n s z)),
ZERO = s\z\z,
reduce (SUCC (SUCC ...(SUCC ZERO)... ))
```

is cubic with the number of SUCC*'s.*

Because of long head-normalisation the actual goal is

```
SUCC = n\s\z\(s (n x\(s x) z)),
ZERO = s\z\z,
reduce a\b\(SUCC
              c\d\(SUCC
                     ...
                     u\v\(SUCC ZERO x\(u x) v)
                     ...
                     x\(c x) d)
              x\(a x) b)
```

The rightmost s *in* (SUCC ...) *is η-expanded because it expects an argument. Applications* (SUCC ...) *expect 2 arguments. Term* ZERO *expects 2 arguments but it is a logical variable. λ-variables* a, c, *and* u *have been introduced by the η-expansion process and are also η-expanded.*

The reduction is shown in figure 3.21. Irrelevant intermediary steps are skipped so that several elementary β-reductions may be necessary to go from one line to the next one. Carets underline what is duplicated during a reduction step, and numbers refer to the following enumeration.

1. *First reductions carelessly nest abstractions created by long head-normalisation.*
2. *They introduce parasitic β-redexes that need a lot of copying. Note that recognising combinators does not help because the terms that are duplicated* are not *combinators.*
3. *Third* SUCC *causes more nesting and parasitic β-redexes that ...*
4. *... cause more copying.*

Note that the amount of copy for reducing each SUCC *is the area of a triangular pattern that grows with the rank of the* SUCC*. Hence the cubic time.*

We observe that the parasitic β-redexes that are created by η-expansion can be reduced using a simpler rule than β-reduction. We call this simpler rule *weak β-reduction* (noted $\overline{\beta}$). It is based on axiom $(\lambda x.(E\ x)\ F) =_{\overline{\beta}} (E\ F)$, where variable x does not occur in E. The interest of $\overline{\beta}$-reduction is that it can be implemented without duplicating the body of the abstraction, E.

Long head-normalisation creates terms $\lambda\overline{x}.(E\ \overline{x})$ where variables $\overline{x}$ do not occur in E. We represent abstractions which are created by η-expansion with a different sort so that these terms are easily spotted. The new sort of abstraction is represented as

```
(mtu SORT_ETA_ABS N+1 x1 ... xN E).
```

Note that the two sorts of abstraction must be considered the same way in every procedure (unification, display) except in β-reduction.

Example 3.6.2
This improvement makes the goal in example 3.6.1 quadratic with the number of SUCC*'s. The reduction is shown in figure 3.22. It performs $\overline{\beta}$-reductions in place of some β-reductions. Hence, reduction does less copying.*

1. *The reduction goes as above ...*
2. *... except that no duplication is required for parasitic β-redexes ($\overline{\beta}$-reduction).*
3. *Again, third* SUCC *nests more abstractions but ...*
4. *... no duplication is required.*

Now, the amount of copy for reducing each SUCC *is the length of a pattern that grows with the rank of the* SUCC*. Hence the quadratic time.*

Among the terms that are η-expanded are the λ-variables. η-expanded λ-variables must be recognised when applying a substitution because it is useless to keep their supplementary binder in the result: $(\lambda z.(x\ z))[t/x] = t$. This preserves the invariant that all applications and constants are in long head-normal form.

This improvement is similar to what procedure TRIV does for spotting trivial pairs. Having distinguished abstractions caused by η-expansion makes it simply easier.

Example 3.6.3
The two improvements make the goal in example 3.6.1 linear with the number of SUCC*'s. The reduction is shown in figure 3.23. The improved reduction does not nest abstractions coming from η-expansion. This produces less β-redexes, hence less reduction steps and less copying.*

1. *The reduction goes as above except that ...*
2. *... η-expanded λ-variables are recognised, so that abstractions are not nested.*
3. *One $\overline{\beta}$-reduction is enough to go to third* SUCC *and ...*
4. *... neither reduction nor duplication is required.*

The cost of reducing each SUCC *is constant. Hence, the linear time.*

3.6.5 A weak substitute for clause indexing

Clause indexing is the exploitation of the clause heads contents for computing more direct clause selection procedures. We have not yet implemented it.

Usually, when control enters a clause that is not the last 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.

We have partially compensated the lack of clause indexing 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 an unknown is substituted, or at the end of unification, if no unknown 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 long head-normal-form to a non-normal form do not count as substitutions of unknowns. So they do not trigger the choice-point creation. The neat effect is that a goal argument is reduced only once before all the attempts to unify it with 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. Note that the brute force solution consisting in reducing a goal before unifying

1. kills lazyness, and
2. does not eliminate the need for normalising during unification because substitutions might build redexes.

Example 3.6.4
In program

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

redex `(s\z\(s(s(s(s(s(s z)))))) x\x 1)` *is reduced only once instead of twice.*

So, delayed creation of choice-point gives a partial solution to a critical problem that appears every time normalisation of terms or awakening of constraints are possible.

Chapter 4

Evaluation of Performances

4.1 The compiler

All that we have presented in the previous sections is implemented in a system called Prolog/Mali. The compiler source amounts to about 6000 λProlog lines. When translated into C, it amounts to 50000 lines ($\approx$ 1 MPPM statement per line). The source of the virtual machine and run-time system is made of 7000 lines of C.

4.1.1 Description of the system

Prolog/Mali implements the complete core of λProlog plus extensions such as a catch/throw escape mechanism, predicates **freeze** and **setof**, and a debugging environment. A DCG expansion mechanism for a typed and higher-order variant of the DCG formalism (*Definite Clause Grammars* [64, 61]) is also part of Prolog/Mali. Since the structure of grammar rules is still that of definite clauses, the extended system still deserves the name DCG. A variant of the DCG formalism following the structure of hereditary Harrop formulas would better be called HHG (*Hereditary Harrop Grammar*) [44].

4.1.1.1 Separate compilation

Prolog/Mali favours *separate compilation* and the building of linkable object files. That the compiler is rather slow is also a strong incentive for using separate compilation. Like for C compilers, compilation is a multi-staged process that can be entered and exited at any stage.

Modularity is managed as in C, with the difference that locality of symbols is the rule rather than the exception.

4.1.1.2 Stand-alone application

Prolog/Mali programs are always compiled in *application mode*. This means that the compiler produces stand-alone executable files. Furthermore, Prolog/Mali applications follow the host-system call/return conventions and use the standard input/output ports.

All Prolog/Mali applications also recognise options of the form `-PM_...`. They may be used for controlling execution: e.g. for specifying memory management when the standard behaviour does not fit the application, or for ordering a monitored execution of the application.

Every Prolog/Mali application should contain in one of its modules a definition of the following form for the predicate `main`:

```
type main
        int -> (list (list int)) -> o.
main  Argc  Argv :-
        ...
```

Argument `Argc` is unified with the number of words in the calling command. Argument `Argv` is unified with the list of these words. If the call to the program is

```
% prog arg1 ... argN
```

the initial goal is

```
  (main  N+1  ["prog","arg1", ...,"argN"])
```

4.1.1.3 Predicates *freeze* and *setof*

Predicate *freeze*

Predicate **freeze** is a transposition of the predicate of PrologII with same name [74, 42]. In PrologII, the semantics of goal

```
(freeze Term Goal)
```

is that `Goal` is not executed as long as `Term` is an *unknown*. In λProlog, `Goal` is not executed as long as `Term` is *flexible*. As we have seen in several occasions, flexible terms must be considered as unknowns.

The versatility of the representation with MALIv06 makes it easy to attach suspended goals to an unknown, and to detach them when the unknown gets substituted. Operationally speaking, to add predicate **freeze** to λProlog is just natural since the basic mechanism must exist for delaying resolution of flexible-flexible pairs.

Predicate *setof*

Predicate **setof** is a transposition of the predicate of Standard Prolog with same name. In Standard Prolog, the semantics of **setof**, which is roughly to accumulate solutions of a goal, is made hairy by the lack of scoping construct. The semantics of goal

```
(setof Pattern Goal List)
```

depends on the run-time occurrence of unknowns in `Pattern` and `Goal`. Unknowns that occur in `Pattern` (when calling **setof**) are called *local*[1]. Unknowns that occur in Goal and are not local are called *global*. When Goal is executed, its solution substitutions produce instances of Pattern. Those giving the same values to the global variables are accumulated in List. When different values are given to global variables, different List's are accumulated and enumerated by backtracking.

That run-time circumstances must be invoked for giving the semantics of predicate **setof** is in contradiction with the objectives of declarative programming.

We transpose the same semantics to λProlog, but we benefit from the scoping constructs of λProlog for making the scopes of the unknowns explicit. Scopes are no more determined at run time. In λProlog, predicate **setof** is declared and used as

```
type setof
        (list A) -> (A -> o) -> o.
... :-
        ... ,
        setof List p\(sigma L1\(...(sigma Ln\
        (   Goal L1 ... Ln p
        ))...)),
        ... .
```

The type declaration and the layout show that **(setof List)** is a quantifier, just like **sigma** and **pi**. That is why we put the list parameter in this unusual position. The local unknowns L1, ..., Ln are all explicitly quantified by existential quantifications. The pattern, p, is scoped by a λ-quantification.

Example 4.1.1
With the classical "drinkers" program [59]

```
kind (individual, beverage)
        type.
type ('Tim', 'Joe')
        individual.
type (tea, milk, beer, wine)
        beverage.
type drinks
        individual -> beverage -> o.
drinks  'Tim'  tea.
drinks  'Tim'  milk.
drinks  'Tim'  beer.
drinks  'Joe'  tea.
drinks  'Joe'  wine.
```

the query

```
query :-
        setof  L  p\(sigma I\(sigma D\
        (   p = (pair I D),
            drinks I D
        ))).
```

gathers all the individual-beverage pairs in unknown L.

The notion of "global variable" must be slightly altered for taking into account logical variables in program clauses. We have seen in section 1.4.3 (example 1.4.3) that an effect of implication in goals is that logical variables may be kept in the program while disappearing from the goal-statement. However, these logical variables are also global variables, even if they do not occur in the goal. They are easy to find in the program continuation, though it is a new overhead.

[1] There is also a special syntax for explicitly tagging unknowns as local.

Example 4.1.2
The following query

```
query :-
        (   p  X  0
        =>  setof  L  x\
            (   p   1   x
            ;   p   2   x
            ),
        ).
```

must succeed twice because it gives two different bindings to unknown X *in the program. In both cases, unknown* L *is substituted by list* [0].

The notation we have chosen is heavier than the Standard Prolog notation, but it is explicit and does not make the actual quantification of a goal depend on a run-time circumstance.

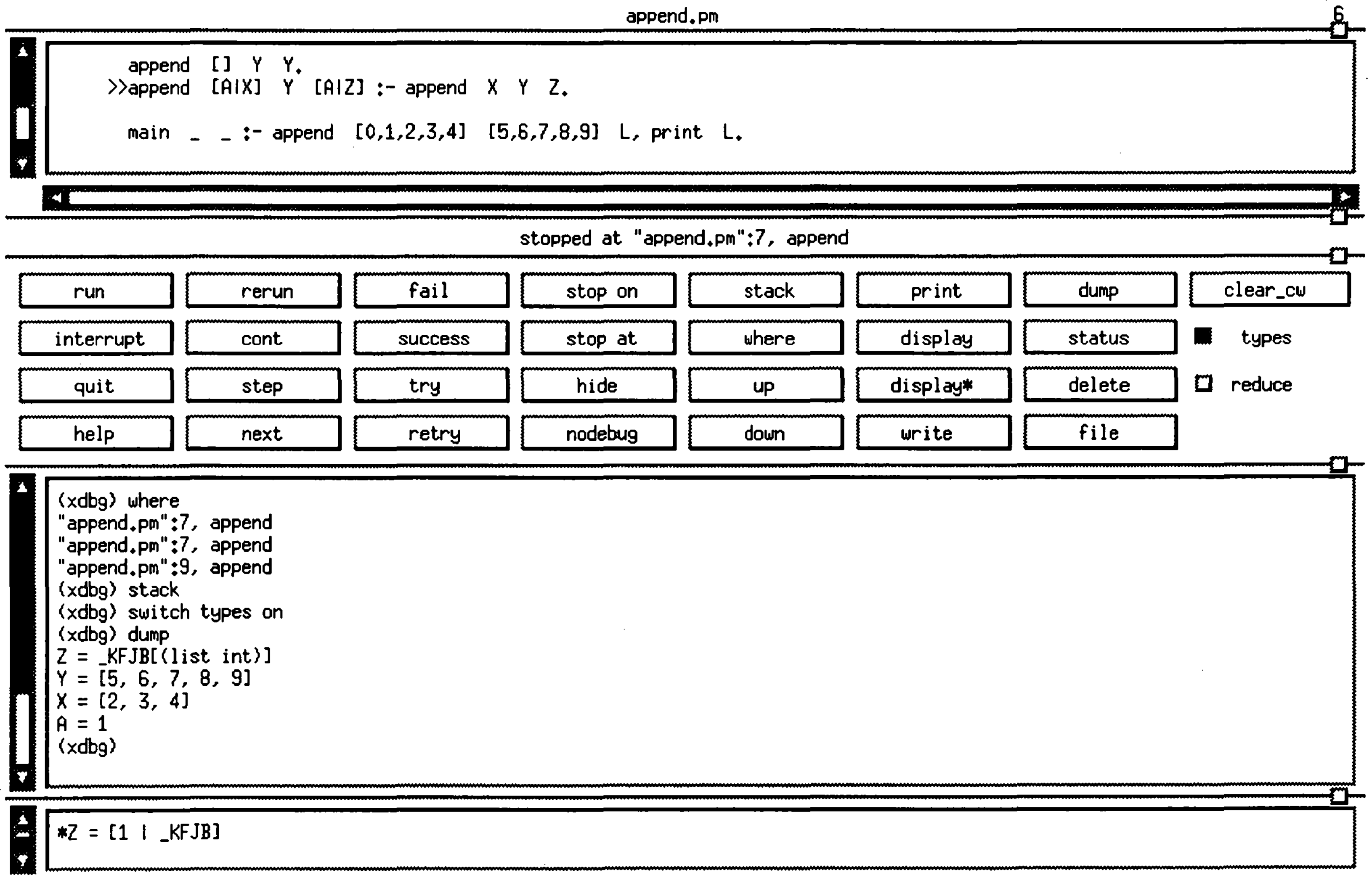
Some problems remain: sorting the instances of the pattern in the list, type unknowns, and delayed unification problems.

- The instances of the pattern are supposed to be put in the list according to a stable, total order on terms. It is easy for ground terms, it becomes quite arbitrary for unknows, and it is completely *ad hoc* for abstractions and flexible applications.
- Type unknowns should also be candidate to be considered as global variables. Definitional genericity should prevent from giving different bindings to the same type unknown in different branches of the search-tree. However, some predicate are definitively not definitionally generic. E.g. built-in predicate **read** has type **A -> o**; imagining its extensional definition in Prolog shows that it is not definitionally generic.
- The instances of the pattern may be qualified by delayed unification problems, which can be interpreted as pending equality constraints. They should probably be part of the accumulated solutions, but are currently forgotten. This problem is common to every logic programming system with constraints or delayed goals.

4.1.1.4 An interactive symbolic debugger

When compiled in debug mode, an application can be monitored using an interactive symbolic debugger. The user can follow the execution step-by-step, or leap over subproofs or leap to break-points. The user can also display terms, the search-stack and the proof-stack.

The debugger interprets the program with the procedural reading of Prolog. The state of the debugger is described by a current goal, a current environment, a call-stack, and a search-stack. The main debugger commands are the following:



append.pm

```
    append  []  Y  Y.
  >>append  [A|X]  Y  [A|Z] :- append  X  Y  Z.

    main  _  _ :- append  [0,1,2,3,4]  [5,6,7,8,9]  L, print  L.
```

stopped at "append.pm":7, append

```
(xdbg) where
"append.pm":7, append
"append.pm":7, append
"append.pm":9, append
(xdbg) stack
(xdbg) switch types on
(xdbg) dump
Z = _KFJB[(list int)]
Y = [5, 6, 7, 8, 9]
X = [2, 3, 4]
A = 1
(xdbg)
```

```
*Z = [1 | _KFJB]
```

Figure 4.1: A snapshot of a debugging session

```
type 'T'                                      /* Terminal string */
        (list A) -> (segment A).
type epsilon                                  /* Condition (empty terminal string) */
        o -> (segment A).
type ('&', ':')                               /* Concatenation and disjunction */
        (segment A) -> (segment A) -> (segment A).
type '-->'                                    /* Derivation */
        (segment A) -> (segment A) -> o.

type 'DCG rule'                               /* Expansion of grammar rules */
        o -> o -> o.
type 'DCG head'                               /* Expansion of rule heads */
        (segment A) -> (segment A) -> o.
type 'DCG pushback'                           /* Pushback list */
        ((segment A) -> (segment A)) -> o.
type 'DCG head'                               /* Expansion of rule heads */
        (segment A) -> (segment A) -> ((segment A) -> (segment A)) -> o.
type 'DCG body'                               /* Expansion of rule bodies */
        (segment A) -> (segment A) -> o.
type 'DCG terminal'                           /* Expansion of terminals */
        (list A) -> (segment A) -> o.
```

Figure 4.2: The DCG translator (declarations)

run *args*, rerun *args* — Starts the execution of a program with arguments *args*. The current goal is the first goal of the first clause of predicate main that matches the argument. The current environment is this clause together with new unknowns. The call-stack is empty. The search-stack contains the goal (main $n+1$ $\overline{arg_n}$) if clauses remain, it is empty otherwise.

fail, succeed — Commits the current goal to fail or succeed without executing it.

step *n*, next *n*, cont *n* — Proceeds on the execution of a program. Each command may succeed or fail. They succeed respectively when the current goal can be rewritten in a clause body, or when it can be solved, or when the end of the program or a breakpoint is reached. Otherwise, they fail.

In case of failure, nothing is changed, and the user can proceed, probably using a finer grained execution command, or can commit the current goal to success or failure.

All these commands may be repeated *n* times.

stop on *p*, stop at *position* — Creates a breakpoint on predicate *p*, or at position *position*.

stack, where *n* — Displays the search-stack, or the *n* most recent levels of the call-stack.

up *n*, down *n* — Moves along the call-stack.

try, retry — Creates an exploratory search-point, and backtracks to it.

print *u*, display *u*, display* *u*, dump — Prints or orders the permanently updated display of the value of unknown *u*. Command display interprets the name of the unknown in the environment of the display; it displays the value of a different unknown in every different environment containing an unknown of that name. Command display* interprets the name of the unknown in the environment in which the display is ordered; it always displays the value of the same unknown. Note that the value may change because of substitutions. Command dump displays the whole current environment.

status, delete *n* — Displays the data-base of breakpoints and display orders. Command delete suppresses entry number *n* from the data-base.

Figure 4.1 shows a snapshot of a debugging session. The graphical debugging environment has four main areas.

1. The topmost area displays the source of the module being executed. The line of the current control point is marked by one of >>, <<, and ><.

   >> The control point is a call to a goal.

   << The control point is a return from a clause.

   >< The control point is a failure.

   The predicate name of the actual control point is also displayed under the topmost area. It is mainly useful when lines contain several goals, which is altogether a bad program lay-out.

2. Second area from the top is a command panel with buttons and switches. It allows to send commands with parameters selected in other areas. In the figure, switch type is on, which means that types of unknown values will be displayed.

3. More elaborated commands can be sent from the third area which also display the history of the debugging session. One can see on the figure the dump of all logical variables of the current clause. Note that logical variable Z is displayed with its type (eighth line).

```
/*      Combinators used for translating connectives. */
#define CONJ       left\right\in\out\(sigma Link\(left in Link, right Link out))
#define DISJ       left\right\in\out\(left in out; right in out)
#define EPSILON    goal\in\out\(goal, in=out)

/*      Definitions of the connectives.
**      They are used at run-time, if a non-terminal is flexible at compile-time.  */
('T' Terminal0)  In  Out :-
        'DCG terminal'  Terminal0  Terminal,
        Terminal  In  Out.
(S1 & S2)  In  Out :-
        CONJ  S1  S2  In  Out.
(S1 : S2)  In  Out :-
        DISJ  S1  S2  In  Out.
(epsilon Goal)  In  Out :-
        EPSILON  Goal  In  Out.

/*      Translates a rule.
**      First strips the rule of its universal quantifications,
**      second splits the rule in a head and a body.  */
'DCG rule'  (pi Rule)  (pi Clause) :-
        pi y\
        (   'DCG rule'  (Rule y)  (Clause y)
        ).
'DCG rule'  (Head0 --> Body0)  (pi In\(pi Out\(Head In Out :- PushBack Body In Out))) :-
        (   'DCG pushback'  PushBack
        =>  'DCG head'  Head0  Head
        ),
        'DCG body'  Body0  Body.

/*      Translates a head and specifies a transformer to be applied to the body.
**      The transformer is either a "conjunctor" or the identity function.  */
'DCG head'  (Head0 & ('T' Terminal0))  Head0 :- !,
        'DCG pushback'  body\(CONJ body in\out\(Terminal out in)),
        'DCG terminal'  Terminal0  Terminal.
'DCG head'  Head0  Head0 :-
        'DCG pushback'  x\x.

/*      Translates a body.
**      Applies a combinator according to the main connective and proceeds.  */
'DCG body'  (Left0 & Right0)  (CONJ Left Right) :- !,
        'DCG body'  Left0  Left,
        'DCG body'  Right0  Right.
'DCG body'  (Left0 : Right0)  (DISJ Left Right) :- !,
        'DCG body'  Left0  Left,
        'DCG body'  Right0  Right.
'DCG body'  (epsilon Goal0)  (EPSILON Goal0) :- !.
'DCG body'  ('T' Terminal0)  Terminal :- !,
        'DCG terminal'  Terminal0  Terminal.
/*      A true non-terminal is ready to get its missing arguments.  */
'DCG body'  NonTerminal0  NonTerminal0.

/*      Translates a terminal.
**      The only part that has to do with lists.  Can be easily changed.  */
'DCG terminal'  Terminal  in\out\(in=(FTerminal out)) :-
        list2flist  Terminal  FTerminal.
```

Figure 4.3: The DCG translator (definitions)

4. Last area from top contains the displays the user has ordered. Here, `*Z` indicates a display of kind `*` of logical variable `Z`.

There is a contradiction in displaying a procedural semantics with a call-stack when the actual execution scheme is based on continuations: continuations give no account of the call history. When compiled in debug mode, every predicate is augmented with special goals that hold the necessary redundancies: call-stack structure, name of unknowns, line numbers, etc.

4.1.1.5 A DCG translation system

The DCG formalism has been adapted to λProlog. The main problem is the strict typing policy of λProlog. In Standard Prolog's DCG, the connective `','` serves both as a logical connective with type `o -> o -> o` in the clause syntax, and as a concatenation connective in the DCG formalism. Furthermore, the DCG formalism accepts strings (denoting terminals), goals (for side-conditions), and true non-terminals in the same syntactic position. One have to find a common type to all these features.

We consider that non-terminals, terminals and side-conditions have type

```
(list A) -> (list A) -> o
```

Though there is no proper means for doing this, we abbreviate it in `(segment A)`. This is the type of a binary relation on strings. It can also be seen as the type of a goal with two arguments missing.

Items are connected with concatenation and disjunction connectives `'&'` and `':'`. Goals are wrapped into term constructor `epsilon` (for *empty* non-terminal), and terminal strings are wrapped into term constructor `'T'`. True non-terminals are not wrapped at all; they are λProlog goals with the input and output arguments missing. The types of these connectives and constructors are given in figure 4.2.

Figure 4.3 shows the main lines of the DCG translator. There is no fiddling with unknowns as in the Standard Prolog version. Adding the missing arguments is done by mere application rather than using the `functor/3` and `arg/3` built-in's. Moreover, unknowns of the generated clauses are correctly represented by explicit quantifications, whereas they are incorrectly represented by unknowns in the Standard Prolog version.

Note how combinators are defined and used in predicates just like we did with function-lists predicates in section 1.5.3.

Translation of terminals (term constructor `'T'`) could have used predicate `'C'` of Standard Prolog. It would not have committed the formalism to work on lists.

4.1.2 A running example (continued): application to predicate *fnrev*

We apply all the mechanisms we have described in chapter 3 to the execution of goal

```
(fnrev z\[1|(z\[2|(z\[3|z] z)] z)] LOut)
```

The unification problem associated with the first parameter of `fnrev` is

```
<z\[1|(z\[2|(z\[3|z] z)] z)],z\[A|(L z)]> .
```

Simplification (done by procedure SIMPL) results in two new problems

```
< z\1, z\A >
```

and

```
< z\(z\[2|(z\[3|z] z)] z), z\(L z) > .
```

The first pair yields substitution `[1 / A]` by imitation. In the second pair, `z\(L z)` is recognised to be η-equivalent to `L`. So, the second pair yields the trivial substitution

```
[z\(z\[2|(z\[3|z] z)] z) / L] .
```

No occurrence-check is required because it is the first occurrence of `L`.

The unification problem associated with the second parameter of `fnrev` is

```
< LOut, z\(RL [A|z]) >
```

or

```
< LOut, z\(RL [1|z]) >
```

if it is treated after the first parameter. It produces the trivial substitution `[z\(RL [A|z]) / LOut]`.

So, one execution step is done in constant time and produces the derived goal

```
(fnrev z\(z\[2|(z\[3|z] z)] z) LOut1)
```

After a β-reduction of the outermost redex of its first parameter, the derived goal has the same profile as the father goal; the next execution step will take the same amount of time. And so on. So, the execution of goal `(fnrev LIn LOut)` with mode `(fnrev + -)` is linear in time with the length of `LIn`.

When the input list is completely β-reduced, `fnrev` is linear for similar reasons. When the list is in the third form (see in section 3.4.1), the access to its first element causes its reduction; then the previous analysis applies.

4.1.3 Application to predicate *append*

We give in this section examples of the actual C code that is produced for two versions of the append predicate. In one version, the predicate is declared monomorphic, whereas in the other it is polymorphic. In the latter case, the produced code has to deal with forgotten types. It results in a few more instructions, and a slower executable code. The speed of the `nrev` predicate using the monomorphic `append` is 26000 Lips on a workstation "sun4/sparc2". It is 23600 Lips with the polymorphic version of `append` (a 10% penalty). The translation takes 5 seconds and produces about 100 lines of C for predicates `append` and `nrev`. Much more lines are actually generated because of the default definitions. The compilation and linking of the C code

```
#       ifdef polymorphic
type append
        (list A) -> (list A) -> (list A) -> o.
#       else  monomorphic
type append
        (list int) -> (list int) -> (list int) -> o.
#       endif

append  []  X  X.
append  [E|X]  Y  [E|Z] :-
        append  X  Y  Z.
```

Figure 4.4: (Poly/Mono)morphic *append*

```
static void PM_c_append()
{ loop:
  SELECT_ENTRY()
  { PM_ENTRY(1L):                                        /*  type A = 128  X = 2  */
    { RECORD_TRY(2L);
      if (
        UN_T_UNK1(REG_TYPE(1), 128) &&                                 /* (1) */
        UNA_NIL(1) && UNA_UNKN(3, 2)) && FINISH_UNIFY())
      { COMMIT_CHOICE(); CONTINUE() } else { FAIL_TRY(); }
    }
    PM_ENTRY(2L):                      /*  type A = 128  E = 127  X = 1  Y = 2  Z = 3  */
    { RECORD_TRUST();
      if (
        UN_T_UNK1(REG_TYPE(1), 128) &&                                 /* (2) */
        UNA_CONS(1, &xx[1]) &&
          UN_UNK1(CONS_SELECT_CAR(xx[1]), 127,
            ( MK_T_UNKN(UNK_SELECT_TYPE(control.robj.unk), 128))) &&   /* (3) */
          UN_UNK1(CONS_SELECT_CDR(xx[1]), 1,
            ( MK_T_APPL1(&PM_k_list, UNK_SELECT_TYPE(control.robj.unk), &xx[2]),
              MK_T_UNKN(T_APPL1_SELECT_ARG(xx[2], 1), 128))) &&        /* (4) */
                                                                       /* (5) */
        UNA_CONS(3, &xx[1]) &&
          UN_UNKN(CONS_SELECT_CAR(xx[1]), 127) &&
          UN_UNK1(CONS_SELECT_CDR(xx[1]), 3,
            ( MK_T_APPL1(&PM_k_list, UNK_SELECT_TYPE(control.robj.unk), &xx[2]),
            MK_T_UNKN(T_APPL1_SELECT_ARG(xx[2], 1), 128)))) &&         /* (6) */
        FINISH_UNIFY())
      { COMMIT_CHOICE(); MK_FIRST_GOAL(&PM__append);
        MK_T_UNKN(REG_TYPE(1), 128);                                   /* (7) */
        LOOP();                                                        /* (8) */
      } else { FAIL_TRUST(); return; }
} } }
```

Figure 4.5: Compilation of the polymorphic *append* (edited)

```
static void PM_c_append()
{ loop:
  SELECT_ENTRY()
  { PM_ENTRY(1L):                                                    /* X = 2 */
    { RECORD_TRY(2L);
      if ( UNA_NIL(1) && UNA_UNKN(3, 2) && FINISH_UNIFY())
      { COMMIT_CHOICE(); CONTINUE(); } else { FAIL_TRY(); }
    }
    PM_ENTRY(2L):                                  /* E = 128 X = 1 Y = 2 Z = 3 */
    { RECORD_TRUST();
      if (
        UNA_CONS(1, &xx[1]) &&
          UN_UNK1(CONS_SELECT_CAR(xx[1]), 128,
            ( MK_T_SYMB(&PM_k_int, UNK_SELECT_TYPE(control.robj.unk)))) &&      /* (1) */
          UN_UNK1(CONS_SELECT_CDR(xx[1]), 1,
            ( MK_T_APPL1(&PM_k_list, UNK_SELECT_TYPE(control.robj.unk), &xx[2]),
              MK_T_SYMB(&PM_k_int, T_APPL1_SELECT_ARG(xx[2], 1)))) &&
        UNA_CONS(3, &xx[1]) &&
          UN_UNKN(CONS_SELECT_CAR(xx[1]), 128) &&
          UN_UNK1(CONS_SELECT_CDR(xx[1]), 3,
            ( MK_T_APPL1(&PM_k_list, UNK_SELECT_TYPE(control.robj.unk), &xx[2]),
              MK_T_SYMB(&PM_k_int, T_APPL1_SELECT_ARG(xx[2], 1))))) &&
        FINISH_UNIFY())
      { COMMIT_CHOICE(); MK_FIRST_GOAL(&PM__append); LOOP(); } else { FAIL_TRUST(); }
} } }
```

Figure 4.6: Compilation of the monomorphic *append* (edited)

takes 2 seconds. The speed of an efficient Standard Prolog compiler (SICStus 0.7) is about 170000 Lips.

Figures 4.5 and 4.6 show the actual generated code. Its commands follows the principles of the MPPM, but are slightly different for dealing with the specifics of λProlog. For instance, command LOOP is almost the same as

```
JUMP_GOAL(&PM__append); goto loop
```

but it has also to deal with awakened constraints and inserted MATCH goals. Command FINISH_UNIFY has to do with delayed unification. For dealing with delayed creation of choice-points, or-control commands are split into commands RECORD_something for recording the delayed or-control operations, and a command COMMIT_CHOICE for executing the current delayed operation.

We do not want to insist too much on a particular intermediate machine. What is important is to specialise and to extend MALI in the style of the MPPM. The intermediate machine is an unstable part of the Prolog/Mali system. It must satisfy two requirements that happen to be contradictory:

- Code for the intermediate machine must be easily generated. This leads to many variants of the commands for fitting the different contexts. E.g. unification commands can be specialised for operating on argument registers (write mode is impossible —the type of unknowns need not be created—, the term is necessarily a combinator, the where parameter is a cell in an indexed array —it is easier to denote the index than the cell—). More specialisation can be achieved if one considers the number of occurrences of unknowns.

  One may object that code generation does not matter because a computer is doing the work. However, this is important when designing the execution scheme, the intermediate machine, and its implementation at the same time.

- The intermediate machine must be simple and orthogonal for making its development safer. This means that the machine adapts to the context through parameterisation rather than through diversification of commands.

Figure 4.5 shows the C code produced for the polymorphic append. It calls for a few remarks.

Line (1) The forgotten type of the head is unified with the forgotten type of the goal. Since, it is always the first occurrence of the type unknown, command UN_T_UNK1 always operates in read mode. The forgotten type is stored in register 128.

Line (2) The forgotten type must be read for every clause. A first improvement would be to produce the code for forgotten types only once because it is the same for every clause. A second improvement is to execute it only once.

Line (3) Every unknown must carry its type. For a first occurrence in read mode, one has only to read the corresponding goal term. If it is an unknown, then it comes with its type, otherwise no type is required. In write mode, the unknown is created and its type too. Since it only operates in write mode, the type sequence is made of make instructions.

Type sequences corresponding to the forgotten types of **append** are not part of the conjunction of unification commands because they should always succeed[2]. In some sense, they operate in two modes, but one of them is trivial and amounts to doing nothing.

In this case, the unknown (E) has the forgotten type of the predicate. So, it is an n-th occurrence of the type stored in register 128.

Line (4) As for line (3), a type may be created in write mode. In the case of unknown X, it is a complex type (list ...).

Line (5) No code is produced for unknown Y. The goal term, whichever it is, remains in register number 2, ready for the recursive call to append.

Note that unknowns X and Z are loaded in the argument registers 1 and 3, to be ready for the recursive call.

Line (6) The type sequence of unknown Z is like the one of unknown X.

Line (7) The forgotten type of the body goal is installed. It is a copy of register 128. This can be improved in the same way as the unknown arguments.

Line (8) Predicate **append** loops without going through the motor. A jump to label loop is hidden in command LOOP.

In the monomorphic case, every instruction produced for dealing with forgotten types is needless.

Line (1) The type of unknown E is perfectly known (int). It is created under the same conditions as above.

Note that monomorphism saves instructions for forgotten types. But it may be at the price of larger type sequences for the unknowns. Suppose that the type of the list elements is (list (list (list (list int)))), then the type sequence for occurrence of E would grow accordingly.

4.2 A running example (continued): performances of function-lists

The function-list technique makes an intensive use of higher-order unification and β-reduction. The logical advantage it has over the difference-list technique is gained at the cost of a sound implementation of β-reduction. In principle, this means duplicating a function every time it is applied to a term. We have shown how to avoid it in many cases. So, the question is

What about performances?

We have observed that it is efficient. We give the results of some experiments with function-lists and theorem proving applications.

We describe time and space behaviours, and then compare with eLP.

4.2.1 Execution times and memory consumption

Table 4.1 shows the run-times of some of the predicates that we have presented in previous sections, and of a built-in predicate, **reduce**, which normalises terms.

Normalisation must be included in the measured times. The different non-normal forms that a function-list is likely to assume are presented in section 3.4.1. The outputs of the two versions of **list2flist** are not only λ-equivalent as expected, they are also identical. Furthermore, the times for normalising the outputs of either version of **list2flist** and for **fnrev** are the same.

The entry for **reduce** shows the times for the complete reduction of the output of **fnrev**. The entries giving the times for **fnrev** are tagged with (n) when its input is normalised, and (nn) when it is the output of **list2flist**. The entries tagged (1) and (2) show the times for the first (with the **pi**) and second versions of **list2flist**.

It can be seen that **reduce** and the two versions of **list2flist** are linear. This is desirable, but should be noted because the computations involved in these predicates are not generally linear. Finally, **fnrev** appears to be linear (even normalisation included).

Times for large lists show a deviation which is explained because memory is almost exhausted and the garbage collector is called more frequently and has more to do.

The programs measured in table 4.1 do not use the output of the tested predicate after its execution (input is used). Terms can be discarded as soon as they are produced. This makes memory management less intrusive in the time measurement, but it gives an optimistic indication on the capacity of the system. Table 4.2 shows for some predicates the size of the largest list they can operate on while using both their input and output after their execution. This corresponds to the most pessimistic memory requirements. Knowing that the measurements are done with 3 mega-bytes of memory indicates the capacity of the system.

4.2.2 Comparison with eLP

We compare Prolog/Mali with eLP (version 0.15), the Lisp based implementation of λProlog. (Ergo Project at Carnegie Mellon University). To be fair, one must say that eLP is interpreted. However, complexity has nothing to do with the implementation technology, and **fnrev** is quadratic on eLP. So, it is deceptive in the sense of the introduction. Table 4.3 shows run-times for eLP. The two versions of **list2flist** are probably linear, but they are the victims of an intrusive garbage-collector. Note also that no predicate can operate on lists longer than 4096, in spite of the garbage-collector

[2]Type sequences corresponding to the forgotten types of terms belong to the conjunction.

List lengths	32	64	128	256	512	1024	2048	4096	8192
nrev	0.19	0.7	2.6	10	41	167	685	2940	13733
drev	0.01	0.02	0.04	0.08	0.17	0.34	0.67	1.3	2.7
frev	0.07	0.13	0.26	0.52	1	2.1	4.1	8.2	17
(n) **fnrev**	0.46	0.91	1.8	3.6	7.2	14.5	29	59	119
(nn) **fnrev**	0.47	0.94	1.9	3.7	7.5	15.5	33	77	322
(1) **list2flist**	0.1	0.2	0.35	0.7	1.4	2.7	5.5	11	22
(2) **list2flist**	0.05	0.09	0.17	0.35	0.69	1.4	2.7	5.5	11
reduce	0.03	0.06	0.12	0.23	0.46	0.91	1.8	3.6	13

Table 4.1: Prolog/Mali run-times (seconds on sun3/60 with 3 Mbytes)

Predicates	**reduce**	(1) **list2flist**	(2) **list2flist**	(nn)**fnrev**	(n) **fnrev**
Max lengths	9000	9000	9000	4000	8000

Table 4.2: List crunching capability of Prolog/Mali (list lengths with 3 Mbytes

List lengths	32	64	128	256	512	1024	2048	4096
nrev	8.6	48.5	⊥	⊥	⊥	⊥	⊥	⊥
fnrev	6.2	20.6	79	338	1600	⊥	⊥	⊥
(1) **list2flist**	.9	1.7	3.8	8.2	20	53	152	⊥
(2) **list2flist**	.5	1	2.2	4.9	10.7	27	89	⊥

Table 4.3: eLP run-times (seconds on sun4 with 14 Mbytes)

List lengths	32	64	128	256	512	1024	2048	4096	8192
Prolog/Mali **nrev**	0.024	0.092	.35	1.4	6.1	26.4	121	552	2450
Prolog/Mali **fnrev**	0.01	0.02	0.05	0.1	0.21	0.61	1.07	2.5	6.4
(eLP) **fnrev**	4.6	15.5	56	240	1107	⊥	⊥	⊥	⊥

Table 4.4: Compared run-times (seconds on sun4/sparc2 with 32 Mbytes)

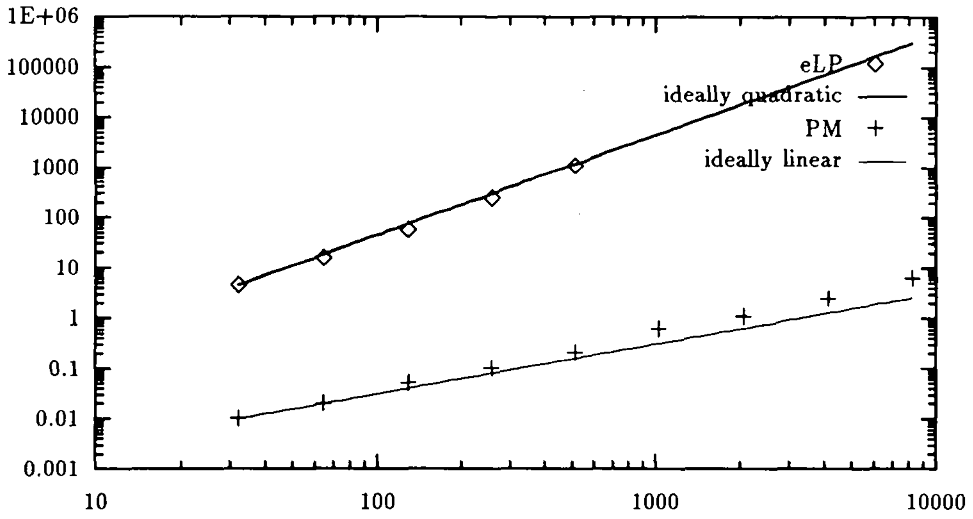


Figure 4.7: Comparison of time complexities for reversing a function-list (list-length × run-times in seconds, log-log scale)

of Lisp and the nearly 5 times bigger memory space. Table 4.4 allows to comparing absolute speeds when executing **fnrev**. The sign $\perp$ means that the computation exhausted memory.

Figure 4.7 shows a graphical insight to the difference between the complexities of Prolog/Mali and eLP for reversing a functional list. Time is given as a function of the length of a 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 latter.

4.3 Performances of theorem proving applications

The bench-marks of section 4.2 were chosen on purpose for exhibiting algorithmic differences. We now compare Prolog/Mali and eLP on applications *that we have not written*. We again compare space and time behaviours.

4.3.1 Memory management

If we consider a wider range of applications, it appears that Prolog/Mali is about fifty times faster than the eLP interpreter (version 0.15 in Common Lisp) and that its memory management is qualitatively better. Prolog/Mali often terminates in much less time than needed by eLP to overflow the memory. As eLP is an interpreter the comparison is partly unfair. However, we believe that it is instructive since it shows that the underlying memory management of Lisp does not help λProlog much. The reason comes from the lack of knowledge of Prolog usefulness logic in Lisp memory management. The muterm of MALI offers the required flexibility and are properly memory managed. This shows that the memory management of MALI is efficient and compatible with a normally fast implementation.

One of the largest application we have tested in both Prolog/Mali and eLP is a theorem prover for sequent calculus improved with a mechanism for cutting redundant subproofs [11]. It involves large data-structures (for example, deduction rules and proofs are encoded in terms). When asked to solve Andrew's problem [63], the eLP version overflows in about forty-five minutes with a ten mega-bytes memory, while the Prolog/Mali version solves it in about eight minutes with one megabyte memory (or less than eleven minutes with 500 kilobytes memory).

Table 4.5 shows times and numbers of garbage collections measured with different memory sizes. It can be observed that Prolog/Mali does not collapse when memory size decreases. With the minimal memory space that allows execution without overflowing, we measure a time shorter than twice the time measured with a large memory.

This validates our giving priority to an efficient memory management. In particular, we believe that muterms help greatly in the improvement of the reduction and unification procedures. They give a way to represent mutable but still backtrackable structures and to ensure an efficient memory management.

Memory sizes	475	480	500	550	600	700	800	1000	2000	10000
run-times	800	700	632	586	565	540	525	515	500	490
numbers of GC	316	302	244	184	149	111	90	65	27	5

Table 4.5: Memory efficiency for Andrew's problem (seconds and Kbytes on sun3/60)

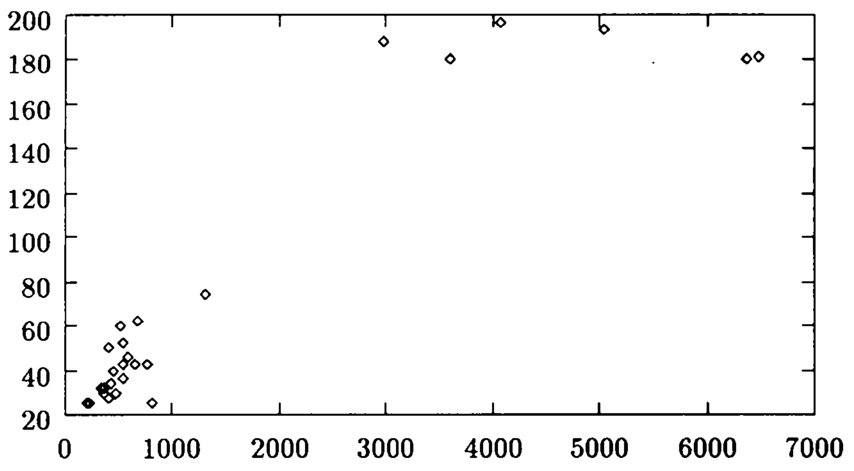


Figure 4.8: Comparison of run-times for executing a tactical theorem prover (run-times in seconds × speed-ups)

4.3.2 Times

We compare Prolog/Mali and eLP on the implementation of an intuitionistic theorem prover written in λProlog. It is an interactive theorem prover which can be driven by the user to use some tacticals. The outcome is a sequent proof or a natural deduction proof.

We measure how much time Prolog/Mali and eLP take for executing the proofs of a set of small problems. The small problems are of the following type[3]:

$[\ (\forall X.\ \neg(prime\ X) \vee (prime\ X))$
$\wedge\ (\forall X.\ (greater\ (f\ X)\ X))$
$\wedge\ (\forall X.\forall Y.\ (divides\ X\ (f\ Y)) \Rightarrow (greater\ X\ Y))$
$\wedge\ (\forall X.\ \neg(prime\ X)$
$\quad \Rightarrow \exists Y.(prime\ Y) \wedge (divides\ Y\ X))\ \ ]$
$\Rightarrow\ (\forall X.\ \exists Y.(greater\ Y\ X) \wedge (prime\ Y))$

Figure 4.8 shows the speed-ups of Prolog/Mali over eLP. It lays between 25 and 240. Every point correspond to a particular problem. Execution times with Prolog/Mali are on the X-axis and the speed-ups (Prolog/Mali on eLP) are given on the Y-axis.

4.4 Juggling with variables

The problem is to transform a term containing notations of free variables into a term in which all these variables are explicitly scoped at the term level. An application of this transformation is to make explicit the implicit quantifications of program clauses.

Example 4.4.1
Term

```
(add (add (unk 1) (unk 1))
     (add (cst 1729) (unk 2))
)
```

where (unk 1) *and* (unk 2) *represent variables and* (cst 1729) *is a constant, is transformed into the term*

```
(all X\(all Y\
(   add (add X X)
        (add (cst 1729) Y)
)))
```

The algorithm operates in a single phase. It uses a continuation passing technique and the following property on predicates p of arity n:

$$\forall F.((p\ (F\ 1)\dots(F\ n)) \Leftrightarrow \forall\overline{X_n}.(p\ \overline{X_n}))$$

The difficulty is to construct a term without knowing the number of abstractions at its top. The property shows that a unique second-order quantification does the job. A unification problem that abstracts the $(F\ i)$'s and installs the X_i's is built during the traversal of the term and solved at the end. The second-order quantification is used to build an intermediary representation such as

[3]It may help to know that f can be interpreted as $\lambda x.(x! + 1)$.

```
kind termT type.

type unk
        int -> termT.
type cst
        int -> termT.
type add
        termT -> termT -> termT.
type all
        (termT -> termT) -> termT.

type generic
        termT -> termT -> o.
type tr
        (int -> termT) -> termT -> termT -> termT -> termT -> (termT -> termT -> o) -> o.
dynamic tr.

/*      Out is In with implicit quantifications explicited.
**      Uses second-order universal variable f for representing quantifications
**      in intermediary representation A.  */
generic  In  Out :-
        pi f\(sigma A\
        (   tr  f  In  A  A  Out  in\out\(in = out)
        )).

/*      tr  F  T1  T2  In  Out  C
**      Given a quantified variable constructor F, T1 is in implicit representation,
**      T2 and In are in intermediary representation, and Out is in explicit representation.
**      In translates to Out if T1 translates to T2 and (C In Out) holds.  */

/*      An implicitly quantified unknown translates to an explicitly quantified unknown.
**      An explicit quantification is added in Out.
**      Proceeds, remembering that the intermediate translation of (unk Unk) is (F Unk)
**      in any further context.  */
tr  F  (unk Unk)  (F Unk)  In  (all Out)  C :-
        (   pi In\(pi Out\(pi C\
            (   tr  F  (unk Unk)  (F Unk)  In  Out  C :-
                    (C  In  Out)
            )))
        =>  (C  In  (Out (F Unk)))
        ).
/*      A node translates to a node. Traverses the subterms in continuation passing style.  */
tr  F  (add T1 T2)  (add S1 S2)  In  Out  C :-
        tr  F  T1  S1  In  Out  in\out\(tr F T2 S2 in out C).
/*      A constant leaf translates to itself.  Proceeds.  */
tr  _F  (cst X)  (cst X)  In  Out  C :-
        (C  In  Out).
```

Figure 4.9: Expliciting implicit abstractions (the program)

Numbers of unknowns	1	2	4	8	16	32	64	128
Prolog/Mali (1)	ϵ	ϵ	ϵ	0.07	0.14	0.53	2.6	13
Prolog/Mali (2)	ϵ	ϵ	0.04	0.1	0.3	1.4	5	29
Prolog/Mali (β)	ϵ	ϵ	ϵ	ϵ	0.03	0.03	0.15	0.5
eLP (1)	ϵ	0.1	0.36	1.7	8.5	59	546	$\perp$
eLP (2)	0.07	0.29	0.96	4.2	24	182	1832	$\perp$

Table 4.6: Expliciting implicit abstractions (run-times in seconds)

```
F\
(   add (add (F 1) (F 1))
        (add (cst 1729) (F 2))
)
```

The program is given in figure 4.9.

Table 4.6 shows the times for the transformation. Time is given as a function of the number of distinct free variables. The two entries for Prolog/Mali and eLP correspond to terms containing 1 and 2 occurrences of every free variable. The third line for Prolog/Mali shows the normalisation times. Remember that the default is to leave terms non-normalised. Symbol ϵ indicates that the time is too small to be measured.

Conclusion and Further Work

On the advantages of λProlog

The promoters of λProlog have mostly shown the logical advantages of λProlog (or any language of a similar brand) over Standard Prolog: a greater subset of predicate logic, a better meta-language for encoding logics, etc. We like to emphasize the importance of λProlog for the Art/Science of programming in logic.

The various scoping constructs of λProlog deal with complex relations that otherwise ought to be left to the programmer. One of them, implication in goals, helps in giving global names to terms in a logical fashion. For instance, we use it for handling options in applications. In Standard Prolog, one must either assert an optional fact or continuously pass options from the query to the predicates that depend on them.

λ-terms help in devising "active" data-structures that operate through β-reduction. It also help in defining macros with an elaborated but safe parameter passing (β-reduction again). In fact, λ-terms can be used at three levels:

Syntactic level: Every potential β-redex is reduced at compile-time. The λ-calculus is used as a luxury macro-programming language for producing first-order logic programs.

It may help in hiding the constructors of datatype. Note that in Standard Prolog, the "do-it-in-unification" style goes against information hiding by making explicit mentions of datatype constructors throughout the programs.

Delayed syntactic level: Not all potential β-redexes are reduced at compile-time, but they are always reduced before entering a unification problem. It is as if a part of the macro-expansion is done at run-time.

Semantic level: Some potential β-redexes enter unification problems. It uses the full power of λ-unification like, for instance, with function-lists.

The distinction between the syntactic levels and the semantic level also applies to constraint logic programming systems (e.g. $\mathcal{CLP}(\mathcal{R})$) in which the syntax allows non-linear constraints though the semantics (the solver) can only cope with linear constraints. The programmer hopes that the non-linear constraints will eventually simplify into linear constraints.

Strong typing, though still controversial, also helps in safer programming. It filters many faulty programs before they are executed. We must admit that the current typing scheme is sometimes painful. Further works need to be done on polymorphism, especially *ad hoc* polymorphism and inclusion polymorphism [16].

Miscellaneous conclusions

We have presented a compiled implementation for the language λProlog. This implementation has been made with the MALI memory. Several lessons can be drawn.

- The use of MALI made the development of this implementation (relatively) easy and fast. This new implementation experiment proves the versatility and the appropriateness for logic programming of MALI. The bad point is that it costs at run-time. The overhead is obvious in the implementation of Standard Prolog, because versatility does not help very much. But it is actually used for non-standard Prolog, and the overhead is probably less detectable.
- We have used list manipulation as a pretext to expose efficiency issues. Primitive operations such as unification and reduction must be carefully designed so as to have a non-deceptive system. The priority given to memory management allows to cope efficiently with large lists.
- We have implemented the λ-calculus primitive operations on a graph-reduction basis. A result of this study is a set of techniques which are the necessary companions of a graph-reduction oriented implementation of λProlog. These techniques are: a lazy outermost β-reduction, the recognition of combinators, and a TRIV that knows about η-equivalence and universal variables. The study also confirms the need for an efficient memory management such as the one that MALI offers. Muterms play a crucial role in the sharing and folding of representations.
- 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 trivial. It appears that the Prolog cut is merely a capture of the failure continuation (a reification) followed by its reinstalment (a reflection). Continuation capture is also used for implementing a catch/throw escape system. All this comes for free by using MALI.

Some general remarks can be done on the representation and processing of typed λ-terms in λProlog.

- Type variables as result types make the arity of some terms unpredictable. So, dynamic η-expansion is required.
- λ-unification and forgotten types require to represent types at run-time.

Forgotten types make dynamic type-checking mandatory, even in the absence of λ-unification. Checking can be limited to forgotten types.

λ-unification is required to know the types of unknowns and constants at run-time. In Prolog/Mali, types of unknown are represented at run-time, and types of constants can be reconstructed given their forgotten types and their result types.

- Full normalisation is not necessary, long head-normal form is enough. This leads to lazy reduction.
- A depth-first strategy for the search of a unifier is not complete. However, it is the usual compromise that leads to a more efficient implementation.
- Detection of combinators avoids many copies and improves sharing. They can be detected easily since all terms constructed by the MATCH procedure are combinators. Moreover, since all instances of combinators are combinators a compiler can usefully detect them in the source program. Every unknown stands for a combinator. Note that this feature has no equivalent in functional programming. This shows that the functional and logical technologies are not merely merged in an implementation of λProlog. They must enhance each other.
- The relation between λ-quantification and $\forall$-quantification must be kept in mind for improving the implementation of both. Though mapping one on the other is logically satisfactory, they both deserve a specialised implementation.

  More generally, it seems to be self-contradictory to hope for a uniform representation while looking for specialising patterns. Sophistication might increase the number of different representation schemes used at the same time. For instance, the current system does a timid step towards explicit substitutions in terms (see procedure TRIV in section 3.5.6.2). Their use could be generalised concurrently with the use of immediate substitutions at β-reduction-time.

Further works

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 no clause indexing. Our current implementation task is to devise a compilation scheme for unification and clause indexing so as to bring the performance level of the first-order part closer to the current state of the art. Note also that the sharing of types should be greatly improved by means similar to those that permit an efficient management of unknowns.

Our implementation of λProlog is not that much slower than Standard Prolog: less than 10 time (≈ 5 on the average) slower than a good implementation of Standard Prolog. It shows that λProlog can come close to Standard Prolog in speed. It gives the hope that the use of λProlog may be generalised.

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 fragment of λProlog has a unitary and decidable unification theory. It is easy to test at run time if a flexible term of λProlog belongs to L_λ, but it could be more efficient to detect that some predicate or some argument is always in L_λ.

Last observation is that the type system deserves further study. It should be studied for itself because the current type system is not flexible enough. E.g. it does not cope for overloading, and does not describe accurately what is going on with predicates **name**, **read** or **=..** . We believe that higher-order types may give the required flexibility. It should also be studied for its interaction with compilation (clause indexing and projection).

Availability of MALIv06 and Prolog/Mali

MALIv06 and Prolog/Mali are registered by APP ("Agence pour la Protection des Programmes", 119 rue de Flandre, 75019 PARIS, FRANCE) under numbers 87-12-005-01 and 92-27-012-00.

MALIv06 and Prolog/Mali are available in source form in an anonymous account at IRISA ("Institut de Recherche en Informatique et Systèmes Aléatoires", Campus Universitaire de Beaulieu, 35042 RENNES Cedex, FRANCE).

```
rlogin irisa.irisa.fr -l anonymous
<your e-mail address>
cd maliv06
...
cd pm
...
```

This facility may be altered or suppressed without notice.

Acknowledgements

We thank Yves Bekkers, Serge Le Huitouze, Pascale Louvet, Barbara Oliver, and Igor Stephan for their reading, commenting, and checking draft versions of this report.

Bibliography

[1] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit substitutions. *J. Functional Programming,* 1(4):375–416, 1991.

[2] 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.

[3] H. Aït-Kaci. *The WAM: A (Real) Tutorial.* Technical Report 5, DEC Paris Research Laboratory, 1990.

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

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

[6] H. Barendregt. *The Lambda Calculus: Its Syntax and Semantics.* Volume 103 of *Studies in logic and the foundations of mathematics,* North-Holland, 1981.

[7] H. Barendregt and K. Hemerik. Types in lambda calculi and programming languages. In N. Jones, editor, *European Symp. on Programming,* pages 1–35, Springer-Verlag, 1990. LNCS 432.

[8] Y. Bekkers, B. Canet, O. Ridoux, and L. Ungaro. MALI: a memory for implementing logic programming languages. In K. Fuchi and M. Nivat, editors, *Programming of Future Generation Computers,* North-Holland, 1988.

[9] 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, Salt Lake City, UT, USA, 1986.

[10] Y. Bekkers, O. Ridoux, and L. Ungaro. Dynamic memory management for sequential logic programming languages. In Y. Bekkers and J. Cohen, editors, *Int. Worshop on Memory Management,* pages 82–102, Springer-Verlag, Saint-Malo, France, 1992. LNCS 637.

[11] C. Belleannée. Improving deduction in a sequent calculus. In *CSCSI90 — OTTAWA, Canadian Artificial Intelligence Conf.,* 1990.

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

[13] P. Brisset and O. Ridoux. The architecture of an implementation of λProlog: Prolog/Mali. In *Workshop on λProlog,* Philadelphia, PA, USA, 1992.

[14] P. Brisset and O. Ridoux. Continuations and λProlog. In *Int. Conf. Logic Programming,* Budapest, Hungary, 1993.

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

[16] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys,* 17(4):471–522, 1985.

[17] A. Church. A formulation of the simple theory of types. *J. Symbolic Logic,* 5(1):56–68, 1940.

[18] A. Colmerauer. Prolog and infinite trees. In K.L. Clark and S-Å. Tärnlund, editors, *Logic Programming,* Academic Press, 1982.

[19] T. Coquand and G. Huet. The calculus of constructions. *Information and Computation,* 76(2/3):95–120, 1988.

[20] N.G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae,* 34:381–392, 1972.

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

[22] F. Fages. *Formes canoniques dans les algèbres booléennes, et application à la démonstration automatique en logique de premier ordre.* Thèse de doctorat, Université de Paris VI, 1983.

[23] 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.

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

[25] 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*, pages 61–80, Springer-Verlag, Berlin, FRG, 1988. LNCS 310.

[26] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, (50):1–102, 1987.

[27] M. Hanus. Horn clause programs with polymorphic types: semantics and resolution. *Theoretical Computer Science*, (89):63–106, 1991. Previously in [28].

[28] M. Hanus. Horn clause programs with polymorphic types: semantics and resolution. In *TAPSOFT'89*, pages 225–240, Springer-Verlag, Barcelona, Spain, 1989. LNCS 352.

[29] 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.

[30] G. Huet. Introduction au λ-calcul typé. In B. Courcelle, editor, *Logique et informatique : une introduction*, pages 137–162, INRIA, 1991.

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

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

[33] R.J.M. Hughes. A novel representation of lists and its application to the function "reverse". *Information Processing Letters*, (22):141–144, 1986.

[34] A. Hui Bon Hoa. Some kind of magic for (a restriction of) L_λ. In *Workshop on λProlog*, Philadelphia, PA, USA, 1992.

[35] J. Jaffar and S. Michaylov. Methodology and implementation of a $\mathcal{CLP}$ system. In J.L. Lassez, editor, *4th Int. Conf. Logic Programming*, MIT Press, 1987.

[36] 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, Paris, France, 1991.

[37] B.W. Kernighan and D.M. Ritchie. *The C Programming Language.* Prentice-Hall, 1978.

[38] A.J. Kfoury, J. Tiuryn, and P. Urzyczyn. The undecidability of the semi-unification problem. *Information and Computation*, 102(1):83–101, 1993.

[39] A.J. Kfoury, J. Tiuryn, and P. Urzyczyn. *The Undecidability of the Semi-Unification Problem.* Technical Report BUCS 89-010, Boston University, 1989. Also published in [38].

[40] 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.

[41] 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.

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

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

[44] S. Le Huitouze, P. Louvet, and O. Ridoux. Logic grammars and λProlog. In *Int. Conf. Logic Programming*, Budapest, Hungary, 1993.

[45] J.W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, Berlin, FRG, 1984.

[46] 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*, Springer-Verlag, New York, Tübingen, FRG, 1989. LNAI 475.

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

[48] D.A. Miller. A proposal for modules in λProlog. In *Workshop on λProlog*, Philadelphia, PA, USA, 1992.

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

[50] D.A. Miller. Unification of simply typed lambda-terms as logic programming. In K. Furukawa, editor, *8th Int. Conf. Logic Programming*, pages 255–269, MIT Press, Paris, France, 1991.

[51] D.A. Miller. Unification under a mixed prefix. *J. Symbolic Computation*, (14):321–358, 1992.

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

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

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

[55] R. Milner. A theory of type polymorphism in programming. *J. Computer and System Sciences*, (17):348–375, 1978.

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

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

[58] 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.

[59] L. Naish. *Negation and Control in Prolog.* Springer-Verlag, 1986. LNCS 238.

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

[61] R. Pareschi and D.A. Miller. Extending definite clause grammars with scoping constructs. In D.H.D. Warren and P. Szeredi, editors, *7th Int. Conf. Logic Programming*, pages 373–389, MIT Press, Jerusalem, Israel, 1990.

[62] L.C. Paulson. Natural deduction as higher-order resolution. *J. Logic Programming*, (3):237–258, 1986.

[63] F.J. Pelletier. Seventy-five problems for testing ATP. *J. Automated Reasoning*, 2:191–216, 1986.

[64] F.C.N. Pereira. Prolog and natural-language analysis: into the third decade. In S. Debray and M. Hermenegildo, editors, *2nd North American Conf. on Logic Programming*, pages 813–832, MIT Press, 1990.

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

[66] Zhenyu Quian. *Unification of Higher-Order Patterns in Linear Time and Space.* Technical Report 5/92, Universität Bremen, Germany, 1992.

[67] G. Revesz. *Lambda-Calculus Combinators and Functional Programming. Cambridge Tracts in Theoretical Computer Science*, Cambridge University Press, 1988.

[68] O. Ridoux. The compilation of λProlog with MALI. In *Logic Programming Winter School and Seminar*, pages 233–283, Brno, Czechoslovakia, 1992.

[69] O. Ridoux. *MALIv06: Tutorial and Reference Manual.* Publication Interne 611, IRISA, 1991.

[70] J.A. Robinson. A machine-oriented logic based on the resolution principle. *JACM*, 12(1):23–41, 1965.

[71] W. Snyder and J. Gallier. Higher-order unification revisited: complete sets of transformations. *J. Symbolic Computation*, (8):101–140, 1989.

[72] L. Sterling and E. Shapiro. *The Art of Prolog.* MIT Press, 1986.

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

[74] M. Van Caneghem. *L'anatomie de PrologII.* InterÉditions, Paris, 1986. Previously in [73].

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

Index

LISTE DES DERNIERES PUBLICATIONS INTERNES PARUES A L'IRISA

PI 680	BRANCHING BISIMULATION FOR CONTEXT-FREE PROCESSES Didier CAUCAL, Dung HUYNH, Lu TIAN Octobre 1992, 36 pages.
PI 681	DEADLOCK MODELS AND GENERAL ALGORITHM FOR DISTRIBUTED DEADLOCK DETECTION Jerzy BRZEZINSKI, Jean-Michel HELARY, Michel RAYNAL Octobre 1992, 26 pages.
PI 683	TARGET TRACKING BY VISUAL SERVOING Aristide S. SANTOS, François CHAUMETTE Octobre 1992, 50 pages.
PI 684	UNE DESCRIPTION LINEAIRE COMPLETE ET IRREDONDANTE DU POLYTOPE ASSOCIE AU PROBLEME DU VOYAGEUR DE COMMERCE ASYMETRIQUE A 6 SOMMETS Reinhardt EULER, Hervé LE VERGE Octobre 1992, 30 pages.
PI 685	MISE EN CORRESPONDANCE DE SEGMENTS DANS UNE SEQUENCE D'IMAGES PAR UNE APPROCHE LOCALE Samia BOUKIR, Patrick BOUTHEMY, François CHAUMETTE, Didier JUVIN Octobre 1992, 30 pages.
PI 686	FROM EQUATIONS TO HARDWARE. TOWARDS THE SYSTEMATIC MAPPING OF ALGORITHMS ONTO PARALLEL ARCHITECTURES François CHAROT, Patrice FRISON, Eric GAUTRIN, Dominique LAVENIER, Patrice QUINTON, Charles WAGNER Octobre 1992, 18 pages.
PI 687	THE COMPILATION OF λ PROLOG Pascal BRISSET, Olivier RIDOUX Novembre 1992, 90 pages.

EDITEUR
INRIA - Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 LE CHESNAY Cedex (France)

ISSN 0249 - 6399

RR_1831