PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1984,
edited by ICOT. © ICOT, 1984

METACONTROL OF LOGIC PROGRAMS IN METALOG

Mehmet DINCBAS - Jean-Pierre LE PAPE

Centre National d'Etudes des Télécommunications
Route de Trégastel
22301 LANNION CEDEX - FRANCE

ABSTRACT

This paper presents a new and efficient implementation of METALOG that provides meta-level control expression facilities for Horn clauses logic programming.

In METALOG, control information, specified in another logic program, is viewed as metaknowledge, expressing how to use the object-level knowledge. The meta-level expression of the control information gives to a user possibilities to intervene on the deduction process, to define his strategies and then to specify his own interpreter. For this reason, METALOG can be considered as a meta-PROLOG.

The two levels of knowledge are expressed in Horn clauses : object-knowledge in clauses and metaknowledge in metaclauses. METALOG provides a set of control-predicates that one must use in metaclauses in order to indicate his strategies and heuristics to the meta-interpreter.

Several examples are given illustrating the powerful control expression facilities concerning almost all aspects of the inference process. We also give, for some well-known problems, the performances (CPU time, number of deduction cycles, etc...) obtained in METALOG and compare them with a standard PROLOG interpreter.

As one can see from these examples, the combination of object and meta-levels gives to the system more natural and powerful control expression facilities than any other technique, and a great efficiency in the execution of programs.

1 INTRODUCTION

The METALOG system has been defined in order to meet the inadequacies of PROLOG as a problem solver (or "inference engine") - notably with respect to the control of deduction process (Dincbas 1980a). The first implementation has been written in PROLOG and initially used as the kernel of PEACE, a prototype expert system for electronic circuit design (Dincbas 1980b). In order to obtain good performance, as much from the point of view of memory requirements as that of execution time, we have undertaken a clearly more efficient implementation in LISP. In the following paper, we shall illustrate this new application of METALOG by giving numerous examples. We shall begin by discussing the problem of deduction control and by showing the weaknesses of PROLOG and the assets of METALOG in this respect. We shall then present the general organization of METALOG and the operation of the meta interpreter. Having presented the metaclauses and their use in the control of the inference process, we shall give instructions for the implementation of METALOG. Finally, we shall conclude by giving numerous examples of the use of the metacontrol expression on some well-known problems showing the power of the control metalanguage.

2 THE PROBLEM OF DEDUCTION CONTROL

The control problem is essential in the problem solvers for the careful direction of the search for a solution, and the avoidance of the exponential explosion of the search space. It concerns all aspects of the inference process -forward execution and backtracking process alike, as well as all problems linked to the management of search space. Within this framework, two types of control can be distinguished.

1) Internal control : implemented in the interpreter in the form of general strategies and heuristics (for example efficient forward execution process, intelligent backtracking, detection of decidable loops, efficient management of the resolution tree, etc...).

2) External control : explicitly described by the user in some formal notation.

The METALOG system attempts to provide a solution particularly to this external control problem by giving the system's user the possibility of expressing his particular strategies and heuristics in a metalanguage, and therefore the ability to directly control the information deduction process and even to describe his own problem solver.

The need for a system like METALOG is felt when PROLOG is used as an inference engine for expert systems or for the problems arising from the plan-generation in robotics. In fact, for problems of this nature, the inadequacies of PROLOG emerge in two ways.

a) Inflexibility and limitations of the internal control : The PROLOG interpreter applies the "resolution principle" and constructs the derivation tree according to a static and predefined strategy (Kowalski 1979b) (Van Emden 1977) :

- choice of the leftmost literal in the resolvent, as literal to be solved ;
- choice of clauses in written order for the resolution of the selected literal ;
- in case of failure, systematic backtracking to the last choice made, without taking into account the cause of failure.

Moreover, the interpreter repeats the basic cycle without control between the cycles :

- no detection of (decidable) infinite loops (partially linked to its static strategy).
- no preventive detection of failures in the resolvents.

b) The absence of possibilities of expression of the external control : The user finds it impossible to express his metaknowledge (strategies, heuristics...) if he does not mix it - in certain particular cases - with his knowledge.

We must note that these inadequacies are not at all inherent to Logic but are linked to a particular implementation.

The assets of METALOG are located :

- on a level of internal control : thanks to an "intelligent backtracking" process which is very easy to implement and cheap to execute (see (Dincbas 1979a) and the map colouring problem in chapter 7).
- and, above all, on a level of external control : expression and consideration of a heuristic control given by the user in a metalanguage in the form of metaknowledge.

In the following work, we will discuss the aspect of "external control" in particular. Different approaches have been proposed for the expression of control in Horn clauses programming. We can quote notably :

- control by annotation of variables : this is the case of IC-PROLOG (Clark and McCabe 1980).
- control by the use of metapredicates in clauses : this is the case of the GELER and DIF predicates in PROLOG-II (Colmerauer 1982).
- control by the use of connectives for literal conjunctions in clauses : this is the case of EPILOG (Porto 1982) (Pereira 1982).
- control completely independant of programs : this is the case of studies carried out especially at CERT/Toulouse (Fahmi 1979) (Gallaire and Lasserre 1979) (Gallaire and Lasserre 1982) (Dincbas 1980a) which must be brought together with the study of metarules for expert systems (Davis 1980) (Davis and Buchanan 1977).

The advantage of the last approach with relation to the first ones, is that control is not confined to one clause but is found in all the search space and enables the expression of general strategies (for a survey and a discussion of these methods see (Gallaire and Lasserre 1982)).

METALOG has adopted this last approach and enables the user to control nearly all the deduction process. For this purpose, the user has two levels of language at his disposal :

- an object-language in which his domain-specific knowledge is expressed.
- a metalanguage in which he expresses his metaknowledge i.e. the instructions and heuristics on the use of his knowledge (control information).

Through this metalanguage, the user of METALOG can :

- either express some semantic information about his particular problem in order to reduce the search space.
- or specify a (more or less general) problem solving strategy.

We believe that the object-language alone, is not sufficient for the expression of the strategies and heuristics mentioned above. In order to express "how to use what we know", we must pass from the "object" level to the "meta" level. We claim that an efficient problem solving process is obtained by combining the object-knowledge with the meta-knowledge.

The knowledge/metaknowledge duality in problem solving (and therefore in expert systems) is similar to the logic/control duality proposed by R. KOWALSKI for the analysis of algorithms ("Algorithm = logic + control") (Kowalski 1979a). The idea underlying these formulations rests in the desire to distinguish information which expresses knowledge from the information which expresses how to use that knowledge ; in other words, to create a distinction between knowledge belonging to theory, and metaknowledge belonging to meta theory. The combination of these two "object" and "meta" levels -knowledge/metaknowledge, and language/metalanguage - gives the system the capacity to express knowledge and solve problems much more extensively and efficiently than the first order logic (for other considerations on this combination, see (Bowen and Kowalski 1982) and (Weyhrauch 1980).

3 GENERAL ORGANIZATION OF METALOG

In METALOG, the two knowledge levels are expressed in the same representation framework. This is the clausal form of the predicate logic restricted to the Horn clauses. In other words :

- knowledge is expressed in clauses, and
- metaknowledge is expressed in meta-clauses.

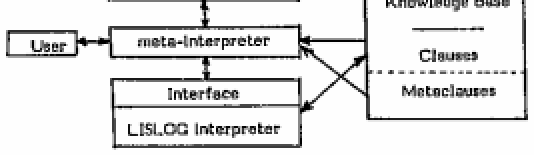
The use of another logic program to specify the control has been first suggested by P. HAYES (Hayes 1973). The approach taken in METALOG improves and generalizes the proposals of H. GALLAIRE & C. LASSERRE (Gallaire and Lasserre 1982) in this direction.

We must recall that clauses (metaclauses respectively) correspond to rules (metarules respectively) in "production systems" (Davis and Buchanan 1977). The predicates and clauses of the object-language are processed as objects (i.e. logic "terms") of the metalanguage. Through the metaclauses, the user has access to the entire set of clauses (he can describe and examine them) as well as the resolution tree (and therefore control of the search space).

The metaclauses are distinguished from the clauses by their head predicate which is a "control predicate", predefined in the system and having semantics known to the interpreter. These control predicates are directly destined for the decision points of the problem solver.

In the new version of METALOG, the notation used for the expression of clauses and meta-clauses, is that of LISLOG - a PROLOG interpreter we have written in LISP and completely embedded in the LISP environment (Bourgault et al. 1984). It has, in particular, the same potential use as evaluable predicates and reducible terms (as in LOGLISP (Robinson and Sibert 1982)).

The METALOG modules



- the meta-interpreter guides the resolution by analysis of metaclauses.
  - it activates requested controls.
  - it constructs the progression strategy (forward or backward), specified by the user.
- the kernel interpreter :
  - applies the progression chosen by the meta interpreter.
  - executes the requested acceptation tests.
- the LISLOG interpreter is activated by the meta-interpreter for the sub-processing of certain problems.

4 THE PRINCIPLES OF OPERATION OF THE META-INTERPRETER

4.1 In the absence of metaclauses

METALOG applies a default strategy identical to that of PROLOG concerning the choice of literals and clauses. In case of failure, METALOG uses an "intelligent" backtracking mechanism which entails backtracking to the most recent resolvent, which is not subsumed[1] by the failed literal, and which presents a non-void list of alternative clauses (Dincbas 1980a). This backtracking mechanism avoids going back towards resolvents destined for failure. It preserves the completeness of the system.

4.2 In the presence of metaclauses

We have seen that metaclauses can be distinguished by the presence of a control predicate known to the system in their head literal.Therefore, at each decision point, the meta-interpreter takes one or many control predicates into consideration. These are mainly :

- for the selection of literals : ACTIVATE and FREEZE.
- for the selection of clauses : CHOOSECLAUSE and INHIBCLAUSE.
- for the choice of a backtracking point in case of failure : BACKFAIL and INHIBACK.
- for requesting a new inference rule "factorisation" : FACTOR.
- for acceptance tests of the resolvent : REJECTGOAL.
- for masking the external control : FINISH.

The objects of the metalanguage are :

- literals, clauses, predicates and objects of the object language.
- the elements composing the resolution tree (resolvent, chosen literal, called clause, literal in failure...).

Example : The metaclause

(ACTIVATE *r *l) <— (totalinst *l)

demonstrates a general strategy which entails choosing a literal (*l) in a resolvent (*r) which is

[1] Let us recall that if C_1 and C_2 are 2 clauses, we can say that C_1 subsumes C_2, if an instance of C_1 is included in C_2, i.e. if a θ substitution exists so that $C_1.\theta \subseteq C_2$ (Kowalski 1979b).

completely instantiated ; "totalinst" might be a predefined predicate of the system, or possibly defined in LISLOG.

Remark : The examples are described in LISLOG's "à la LISP" syntax. In addition variables are prefixed by a "*" (Bourgault et al. 1984).

Analysis of metaclauses

The meta-interpreter investigates metaclauses with respect to the decision point attained and calls them according to their written order in the metaknowledge base. The call of a metaclause is made by subsumption[2] on the arguments of the control predicate. Suppose a metaclause of "action" type

("action" a_1 a_2 ... a_k) <—— $cond_1$ $cond_2$ ... $cond_n$

using the control predicate

("action" t_1 t_2 ... t_k)

where t_i designates a type of object (current resolvent, chosen literal, called clause, ...) known to the system.

For example for (ACTIVATE t_1 t_2)

t_1 refers to the current resolvent and t_2 to a literal candidate for selection i.e. a literal of the current resolvent.

The meta interpreter :

- searches a set of objects O_i (i = 1,k) of type t_i (i = 1,k) which are subsumed entirely by arguments a_i (i = 1,k).
- in the case of success of this subsumption, the evaluation of conditions ($cond_1$ ... $cond_n$) is completely subprocessed in LISLOG.
- in the case of a success in this evaluation in LISLOG, the decision associated with the metaclause is taken.

5 METACLAUSES

Metaclauses enable the specification of an external control - i.e. the expression by the user of his own resolution strategies and heuristics. They ensure the introduction of metaknowledge with the following advantages :

- complete separation between the expression of metaknowledge and that of knowledge : a modification in the set of metaclauses will lead to a completely different program execution.
- the possibility of expressing a very general or very localized control thanks to the subsumption mechanism.

In general, two types of control predicates are available at each decision point of the deduction cycle :

- an activation predicate enabling the activation of a priority decision.
- an inhibition predicate enabling the refusal of the decision by default of the meta interpreter and therefore the requesting of a new decision by default.

5.1 Selection of Literals

* (ACTIVATE r l) <—— condition

where condition is in the form $cond_1$ $cond_2$ ... $cond_n$ $n \geqslant 0$

rôle : To describe a selection strategy of a literal (l) in the current resolvent (r) (the argument corresponding to the object concerned, is shown in brackets).

Examples :

1 (ACTIVATE *r (on A *x)) <——

expresses a particular strategy : unconditional activation of a literal subsumed by (on A *x) in the current resolvent.

2 (ACTIVATE (*p . *q) *p) <——

expresses the PROLOG general strategy (selection of the leftmost literal).

* (FREEZE r l) <—— condition

rôle : To describe a strategy freezing a literal (l) in the current resolvent (r).

Examples :

1 (FREEZE *r (on *x *y)) <—— (var *x)

to freeze the selection of a literal subsumed by (on *x *y) if its first argument is a variable.

2 (FREEZE *r *l) <—— (noninst *l)

general strategy which entails freezing in a resolvent, every literal containing variables.

Comment 1 : The choice of literal in a resolvent is exclusive : it cannot be reconsidered during backtracking.

Comment 2 : These two types of metaclauses lead to the definition of a coroutine mechanism.

5.2 Selection of clauses

* (CHOOSECLAUSE p l) <—— condition

[2] Note that while unification is very well adapted to computation, subsumption (roughly saying "pattern-matching") is more suitable to designate, to select and to manipulate objects (literals, clauses, resolvents, ...).

rôle : To give first choice to a clause whose body (1) is characterized by the metaclause, for the resolution of the selected literal (p).

Example :

(CHOOSECLAUSE *p nil) <—–

expresses the "unit preference" strategy, i.e. for a selected literal p, invocation of unit clauses as first choices.

* (INHIBCLAUSE p 1) <—– condition

rôle : To inhibit the choice of clauses whose body (1) is characterized by the metaclause for the resolution of the selected literal (p).

Examples :

1 (INHIBCLAUSE (toto 5 *x) *1) <—–
(in (titi *y) *1)

indicates to the system to inhibit any clause which contains in its body *l a literal of the form (titi *y) whenever the selected literal p is subsumed by (toto 5 *x).

Comment : These two types of metaclauses are of particular interest for expert systems ; they, in fact, allow content-directed procedure invocation (Davis and Buchanan 1977).

2 Definition of negation in METALOG

(non *x) <—–
(INHIBCLAUSE (non *x) nil) <—– (meta *x)

If *x is proved in METALOG (the system predicate "meta" allows to call METALOG from LISLOG) then the first clause is inhibited and therefore the literal (non *x) is failed. Else (non *x) succeeds.

Remark : see section 5 for another use of the predicate "meta" and another definition of negation under METALOG.

5.3 Choice of a backtracking point in case of failure

* (BACKFAIL p r) <—– condition

rôle : To choose a backtracking resolvent (r) in case of failure on a literal (p).

Example :

(BACKFAIL *p *r) <—– (not ((in $*p_1$ *r)
(subsume *p $*p_1$)))

expresses a general strategy equivalent to the intelligent backtrack mechanism executed by default : choice of a backtracking resolvent not containing a literal subsumed by the failed literal (see the map colouring problem).

* (INHIBACK p r) <—– condition

rôle : To inhibit the choice of a backtracking resolvent (r) in case of failure on a literal (p).

Example :

(INHIBACK *p *r) <—– (in $*p_1$ *r)(subsume $*p_1$ *p)
(not $*p_1$)

expresses a general backtracking strategy with generalization of failure : in the event of failure on a literal, go back to a backtracking resolvent, not containing an unprovable generalization of the failed literal.

5.4 Request for resolution by factorisation

* (FACTOR r l_1 l_2) <—– condition

rôle : To request a supplementary inference by

factorisation with another literal (l_2) for the chosen literal (l_1) in the current resolvent (r).

In the case of success of this metaclause, the action undertaken consists in unifying the two (l_1 and l_2) literals, and in generating a new resolvent from which the second (l_2) literal is eliminated. This is a very elegant (and efficient) way to eliminate the repeated appearance (and therefore execution) of some subproblems during a problem solving process.

Examples :

1 (FACTOR *r (fibo *n *x) (fibo *n *y)) <—–

factorisation of two literals concerning the "fibo" predicate, and having the same first argument.

2 (FACTOR *r $*l_1$ $*l_2$) <—– (totalinst $*l_1$)
(totalinst $*l_2$)
(equal $*l_1$ $*l_2$)

general factorisation strategy of two identical literals.

5.5 Masking of external control

* (FINISH 1) <—– condition

rôle : Assignment of the resolution of the chosen literal (1) to LISLOG.

Example : Processing of negation under METALOG : "mnot"

(FREEZE *r (mnot *x)) <—– (noninst *x)
(FINISH (mnot *x)) <—–

(mnot *x) <—– (meta *x) (slash) (fail)
(mnot *x) <—–

Remark : The predicate "meta" permits to create a new search space in METALOG. Both "FINISH" and "meta" allow to switch from one search space to another and therefore to localize the control (see further the implementation issues).

$$\begin{tikzcd}\text{LISLOG} \arrow[r, bend left, "\text{"meta"}"] & \text{METALOG} \arrow[l, bend left, "\text{"FINISH"}"]\end{tikzcd}$$

5.6 The acceptation tests of resolvents

(REJECTGOAL n r) <—— condition

rôle : Rejection of the current resolvent (r) of depth (n).
Conditions can refer to various types of tests concerning the current resolvent.

Example : Detection of logic loops (Dincbas 1980a) :
(REJECTGOAL *n *r) <—— (resolvent *n_1 *r_1)
(variant *r *r_1)

rejection of a candidate resolvent *r if there is a previous resolvent *r_1, in the proof tree, which is a variant (identical modulo variable names) of *r. Note that (resolvent n_1 r_1) is a system predicate of METALOG restoring all resolvents r_1 of the proof tree beginning with the most recent.

6 SPECIFIC ASPECTS OF THE METALOG IMPLEMENTATION

In addition to usual difficulties encountered in the implementation of PROLOG-like interpreters, new problems arise when implementing METALOG ; among those the four following points are the essential ones.

6.1 Internal representation of objects

The basic principle adopted is structure-sharing. The concepts (term, literal) of the object language are represented in the system by a (s.v) dotted pair where :

- s is a pointer towards a structure (or skeleton) of the knowledge base or the goal to be solved.

- v is a pointer towards an instantiation vector which defines a particular occurence of the skeleton ; for each bound variable of the skeleton, v defines the associated link i.e. a new object of the form (s'.v').

The metalanguage specification requires the ability to handle new concepts : clauses, current resolvent, previous resolvents, chosen literal, literal in failure... The implementation of some of these concepts within the framework of structure sharing, have led us to the extension of the object notion on the internal representation level and to the extension of the associated processing mechanisms (particularly, the unification).

a) Resolvents

The manipulation of resolvents is a frequent operation in metaclauses. Resolvents (or lists of literals) are not generated in the (s.v) form i.e. a homogeneous skeleton with regard to an instantiation vector, but they are naturally represented by a heterogeneous structure like : $((s_1.v_1)\ (s_2.v_2)\ ...\ (s_k.v_k))$. To avoid the copying of resolvents in the form of homogeneous structures, a mechanism for recognition and processing of objects of resolvent type has been introduced into the system.

b) Deferred objects

The metaclauses not only refer to objects of the current resolution step, but also objects of a past resolution step (the BACKFAIL and INHIBACK cases, in particular). In order to manipulate these objects, we would have to :

- either memorize the copies of these objects at each resolution step.
- or memorize copies of instantiation vectors at each step.

To avoid costly copies, we have preferred to develop an incremental construction technique for vectors, which associated with a deferred access mechanism, enables the state of the vector in whatever resolution step to be retrieved ; this technique enables too, if necessary, a new (temporary) extension of the vector from its deferred state.

Another object type appears, therefore, in internal representation : deferred objects. On first encounter with such an object, the deferred access mechanism retrieves the deferred state of the vector and links it with the deferred object.

Four classes of objects have thus been observed in internal representation :

- ordinary objects (term, literal, clause) ;
- resolvent objects ;
- deferred ordinary objects ;
- deferred resolvent objects.

6.2 Management of alternatives

The ability to express a strategy to select the clauses within a subprogram, requires to revise the classical management of choice alternatives in the form of sequential access list. METALOG needs to memorize two types of information in every choice node :

- a dynamic chaining of choice alternatives ;

- a sequential access list of alternatives of CHOOSECLAUSE type metaclauses being waiting in this node ; indeed they are the only metaclauses whose exploration can be activated again during backtracking.

6.3 Mechanism of subsumption

The analysis of metaclauses uses subsumption ; so the corresponding mechanism must have been implemented and extended to the new classes of objects distinguished in the internal representation.

6.4 Overlapping of LISLOG and METALOG search spaces

A resolution under METALOG requires the construction of multiple search spaces :

- the main search space concerning the goal to be solved ;
- a succession of local search spaces corresponding to the analysis of metaclauses conditions ; they are developed at the request of metacontrol and, in general, from the current context of the main search space ; the life of a search space of this type lasts the time of analysis of the concerned metaclause body.

Furthermore, every one of these spaces can be constructed by overlapping, at the option of the user, two types of search subspaces :

- the METALOG type subspaces (with active metacontrol) introduced under LISLOG by the system predicate "meta",
- the LISLOG type subspaces (without metacontrol) introduced under METALOG by the "FINISH" control predicate.

To manage this double complexity in the search, the system integrates mechanisms for switching, for interfacing and for overlapping the control structures associated with the two types of search ; indeed, the transfer between searching spaces must be achieved during forward execution process as during backtracking.

7 USE OF METALOG : EXAMPLES AND PERFORMANCES

Preliminary remarks :

Some abbreviations will be used all along this section in order to simplify the presentation of results :

t	will point out the execution time (measured under MACLISP/MULTICS) expressed in seconds unless otherwise stated ;
n	will point out the number of deduction cycles (or inferences) ;
b	will point out the number of backtracks executed during the resolution ;

The L, M marks will refer to resolutions under LISLOG and METALOG respectively.

If necessary, the M1, M2 ... notation will enable to distinguish several cases of experimentations under METALOG. The results marked by the "*" symbol are estimations of the real values (sometimes with an unlimited memory space). The examples will be described in the LISLOG parenthesized syntax ; they use some system predicates of LISLOG :

(var arg) tests whether arg is a variable ;

(const arg) tests whether arg is a constant ;

(lispeval lisp-expression arg) tries to unify the valuation result of the lisp-expression with the term arg.

Another facility of LISLOG (and METALOG) will be used : the notion of valuable term (reducible term) syntactically introduced by the "&" character. When meeting such a term, the unification mechanism calls LISP to evaluate the term, then carries on the processing, using the value returned by LISP (see (Bourgault et al. 1984)).

7.1 Naïve reverse : "naïve" inversion of a list

Knowledge base :

- clauses :

```
(revn nil nil) <——
(revn (*x . *y) *z) <—— (revn *y *t)
                         (conc *t (*x) *z)

(conc nil *v *v) <——
(conc (*t . *u) *v (*t . *w)) <—— (conc *u *v *w)
```

- metaclauses : two types of metacontrol have been separately tried on this example :

```
M1 : (FREEZE *r (revn *x *y)) <—— (var *x)
M2 : (ACTIVATE *r (conc . *x)) <——
```

M1 and M2 introduce performances which are particularly interesting in the case of the problem "in reverse" i.e. for a problem of form (revn *x (a b c)) : computation of the first argument, knowing the second. M1 asks for freezing the choice of a literal of (revn *x *y) type in a resolvent of *r type if its first argument is a variable. M2 compels the activation of a literal with conc predicate name in a resolvent of *r type.

Number of deduction cycles : (inversion of a list of l length)

without metacontrol	n v $l^3/6$
with FREEZE or ACTIVATE	n v $l^2/2$

Steps of resolution with FREEZE or ACTIVATE : (the literal chosen by METALOG is underlined).

```
                                                  value of *x

1. (revn *x (a b c))                              *x
2. (revn *y1 *t1) (conc *t1 (*x1) (a b c))        (*x1 . *y1)
3. (revn *y1 (a.*u2)) (conc *u2 (*x1) (b c))
4. (revn *y1 (a b. *u3)) (conc *u3 (*x1) (c))
5. (revn *y1 (a b))                               (c . *y1)
6. (revn *y2 *t2) (conc *t2 (*x2) (a b))          (c *x2 . *y2)
7. (revn *y2 (a.*u6)) ((conc *u6 (*x2) (b))
8. (revn *y2 (a))                                 (c b . *y2)
9. (revn *y3 *t3) (conc *t3 (*x3) (a))            (c b *x3 . *y3)
10.(revn *y3 nil)                                 (c b a . *y3)
11.( )                                            (c b a)
```

Performances

l lengtht	10	20	30	50	100
t_L	0.42s	2.7s	9.0s	39.9s	5mn13s
t_{M1}	0.38s	1.3s	2.9s	7.5s	30.1s
t_{M2}	0.27s	0.94s	2.0s	5.5s	22.0s
t_L/t_{M1}	1.1	2.1	3.1	5.3	10.4
t_L/t_{M2}	1.6	2.9	4.5	7.3	14.2
n_L	231	1 561	4 991	22 151	171 801
n_M	66	231	496	1 326	5 151
n_L/n_M	3.5	6.8	10.1	16.7	33.4
b_L	45	190	435	1 225	4 950
b_M	0	0	0	0	0

Comment : This example shows the sensitivity of the performances with respect to the complexity of the metacontrol ; indeed, with FREEZE, the resolution of the metaclause body requires one LISLOG cycle of deduction ; so, for an execution requiring n METALOG cycles, the metacontrol expressed with FREEZE will require n LISLOG cycles ; this explains the variation of performances (in terms of CPU time) between the two types of metacontrol.

As a general rule, the simpler the metacontrol will be carried out, the nearer the ratio t_L/t_M will be of its n_L/n_M ideal value.

7.2 The k-queens problem

This concerns the placing of k queens on a chessboard of k x k squares in such a way that they are not taken. The problem is deliberately expressed simply, to give proof of the advantage of external control.

The user enters his problem in the form of (queen (1 2 ... k) *y) where (1 2 ... k) is the list of line numbers of the chessboard. The search for solutions by the program simply consists in generating one by one the possible *y permutations of this list of numbers, in constructing each time the corresponding list of positions (p number-of-line number-of-column) and in checking each time the coherence of the proposed positions. The check affects only the diagonal tests because the mechanism of positions generation by permutation avoids implicitly the placing of two queens on the same line or the same column.

Knowledge base :

- clauses :

```
(queen *x *y) <——    (perm *x *y)
                     (pair *x *y *z)
                     (safe *z)
(perm nil nil) <——
(perm *x (*u . *v)) <——    (del *u *x *w)
                           (perm *w *v)
(del *x (*x . *y) *y) <——
(del *u (*x . *y) (*x . *v)) <—— (del *u *y *v)
(pair nil nil nil) <——

(pair (*x . *y) (*z . *t) ((p *x *z) . *s)) <—— (pair *y *t *s)
(safe nil) <——
(safe (*x . *y)) <——  (check *x *y)
                      (safe *y)
(check *x nil) <——
(check *x (*y . *z)) <——   (nodiag *x *y)
                           (check *x *z)

(nodiag (p *x *y) (p *z *t)) <—— (absmoins *x *z *u)
                                 (absmoins *y *t *v)
                                 (dif *u *v)

(absmoins *x *z *u) <—— (lispeval (abs (- *x *z)) *u)

(dif *x *x) <——(slash)
               (fail)
(dif *x *y) <——
```

- metaclauses : two sets of metaclauses have been tested separately, the first leading to better performances because it generates a minimum of resolvents.

first set of metaclauses : M1

```
(FINISH (nodiag . *x))
(ACTIVATE *r (nodiag . *x))
(ACTIVATE *r (pair *x (*y . *z) *t)) <——
                              (const *y)
(ACTIVATE *r (check *x (*y . *z)))
(ACTIVATE *r (safe (*x . *y)))
```

These metaclauses will enable the activation of diagonal tests before construction of a complete permutation. Tests are activated each time a new couple of positions is proposed by perm. In the event of failure on these tests, all permutations coming from the partial permutation in failure, are automatically eliminated from the searching process.

second set of metaclauses : M2

```
(FINISH (nodiag . *x)) <——
(BACKFAIL (nodiag *p1 (p *l2 *c2)) ((del *u
                        (*c2 . *y) *v) . *r)) <——
```

In this second set, a complete permutation is allowed to arise, but when diagonal tests fail because of a (p1, p2) couple of positions, the BACKFAIL metaclause enables backtracking to the resolvent which caused the choice of one of the positions of the failing couple ; in this way, a whole class of permutations destined for failure are eliminated from the searching process. More precisely, we can see that the metaclause asks explicitly for the changing of p2, because the generation of p2 appears during resolution after the generation of p1.

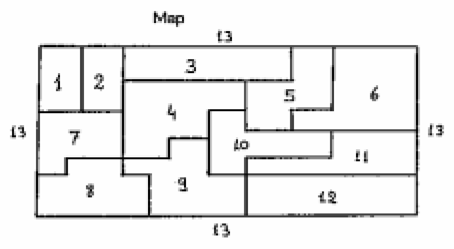
Performances : search for the first queen solution

k	4	6	8
t_L	0.58s	12.6s	4mn18s
t_{M1}	0.90s	6.0s	34.1s
t_{M2}	0.67s	7.2s	1mn5s
t_L/t_{M1}	0.64	2.1	7.6
t_L/t_{M2}	0.87	1.8	4.0
n_L	376	8 446	173 553
n_{M1}	219	1 519	8 965
n_{M2}	299	3 494	32 180
n_L/n_{M1}	1.7	5.6	19.4
n_L/n_{M2}	1.3	2.4	5.4
b_L	49	950	14 693
b_{M1}	11	71	328
b_{M2}	12	82	487

7.3 A map colouring problem

This concerns the colouring of a map with at the most 4 colours in such a way that two adjacent areas do not have the same colour (example taken from (Bruynooghe and Pereira 1983)).

Knowledge base :



Knowledge base :

```
(next blue yellow) <——        (next red blue) <——
(next blue red) <——           (next red yellow) <——
(next blue green) <——         (next red green) <——
(next yellow blue) <——        (next green blue) <——
(next yellow red) <——         (next green yellow) <——
(next yellow green) <——       (next green red) <——

(colouring *1 *2 *3 *4 *5 *6 *7 *8 *9 *10 *11 *12 *13) <——
   (next *1 *2) (next *2 *3) (next *3 *4) (next *4 *5) (next *5 *6)
   (next *6 *11) (next *11 *12) (next *12 *13) (next *9 *13)
   (next *9 *10) (next *4 *10) (next *4 *7) (next *7 *8)
   (next *2 *7) (next *6 *10) (next *2 *13) (next *6 *13)
   (next *8 *13) (next *2 *4) (next *4 *9) (next *3 *5)
   (next *8 *9) (next *1 *13) (next *3 *13) (next *5 *13)
   (next *7 *13) (next *11 *13) (next *9 *12) (next *5 *10)
   (next *10 *13) (next *1 *7)
```

Comparison of three types of execution :

- L execution under LISLOG.
- M1 execution under METALOG without metaclause : the gain here is obtained thanks to the intelligent backtrack mechanism implemented in METALOG : in the event of failure on a l literal, backtrack to a resolvent not containing an instance of l.
- M2 execution under METALOG with addition of the metaclause :

  (ACTIVATE *r (next *C1 * C2)) <—— (const *C1) (const *C2)

  As soon as a literal appears which is completely instantiated in the resolvent, the metaclause enables the activation of this literal which has the effect of accelerating the detection of further failures.

Performances : search for the first solution.

t_L	8 mn 32 s	n_L	94 957	b_L	89 831
t_{M1}	1.2 s	n_{M1}	133	b_{M1}	10
t_{M2}	1.2 s	n_{M2}	44	b_{M2}	8
t_L/t_{M1}	427	n_L/n_{M1}	714	b_L/b_{M1}	8 983
t_L/t_{M2}	427	n_L/n_{M2}	2 158	b_L/b_{M2}	11 229

7.4 Fibonacci function

Knowledge base :

```
- clauses :(fibo 0 0) <——
           (fibo 1 1) <——
           (fibo *n *m) <—— (fibo (& - *n 1) *m1)
                            (fibo (& - *n 2) *m2)
                            (equal (&+*m1 *m2) *m)
           (equal *x *x) <——
```

- metaclause :

(FACTOR *r (fibo *n *x) (fibo *n *y)) <——

This metaclause enables the eventual factorisation of two literals beginning with the "fibo" predicate and having the same first argument, to be requested in a *r resolvent (factorisation = unification of the two literals and elimination of the second of the resolvent). The double recursivity of the function is transformed into simple recursivity during execution.

First steps of resolution :

```
                1. (fibo 20 *x)

                2. (fibo 19 *x1) (fibo 18 *x2) (equal (& + *x1 *x2) *x)

        ┌────── 3. (fibo 18 *x3) (fibo 17 *x4) (equal (& + *x3 *x4) *x1)
        │
     FACTOR                      (fibo 18 *x2) (equal (& + *x1 *x2) *x)
        │
        └─────► 4. (fibo 18 *x2) (fibo 17 *x4) (equal (& + *x2 *x4) *x1)
                                               (equal (& + *x1 *x2) *x)

        ┌────── 5. (fibo 17 *x5) (fibo 16 *x6) (equal (& + *x5 *x6) *x2)
        │                        (fibo 17 *x4) (equal (& + *x2 *x4) *x1)
        │                                      (equal (& + *x1 *x2) *x)
     FACTOR
        └─────► 6. (fibo 17 *x4) (fibo 16 *x6) (equal (& + *x4 *x6) *x2)
                                               (equal (& + *x2 *x4) *x1)
                                               (equal (& + *x1 *x2) *x)
```

Performances

k	10	15	20	50	100
t_L	0.51s	5.3s	59s	3 years 11 months*	10^{11} years*
t_M	0.15s	0.20s	0.27s	0.68s	1.35s
t_L/t_M	3.4	26.5	222	1.8×10^{8}*	2.5×10^{18}*
n_L	265	2 959	32 836	6.1×10^{10}*	1.7×10^{21}*
n_M	28	43	58	148	298
n_L/n_M	9.5	69	566	4.1×10^{8}*	5.7×10^{18}*

8 CONCLUSION

We must remember that the kernel of METALOG interpreter is a theorem prover for Horn clauses using the LUSH - resolution without restriction (Van Emden 1977). Through the control metalanguage, the user can define his resolution strategy and in certain cases even add a new inference rule (as, for example, "factorisation"). Therefore, thanks to a powerful set of metaclauses, the user can directly intervene on the process of information deduction, and even specify his own interpreter. This is why we consider METALOG as a "meta PROLOG".

In the new implementation of METALOG, we have emphasized performance, as much from the point of view of running time as that of memory requirements. Therefore, the technique we have developed for the implementation of "deferred objects" avoids the copying of resolvents which leads to a considerable gain in space. Contrary to the first version in which METALOG was written in PROLOG and in which there were, therefore, two successive levels of interpretation, in the new implementation, the metacontrol module is directly implemented in the interpreter leading to a distinct acceleration in execution.

We believe that the combination of the "object" and "meta" levels (knowledge/metaknowledge and language/metalanguage) gives to a system more natural and powerful control expression facilities than any other technique. If the implementation of such a system is well done, one can get great efficiency in the execution of his problem, as illustrated by the examples in this paper.

REFERENCES

Bourgault S., Dincbas M., Le Pape J.P., Manuel LISLOG. CNET, Note Technique NT/LAA/SLC/159, Janvier 1984.

Bowen K., Kowalski R., "Amalgating language and metalanguage in logic programming". Logic Programming, Academic Press, 1982, p. 153-172.

Bruynooghe M., Pereira L.M., Deduction revision by intelligent backtracking. Univ. Nova de Lisboa, Dept. de Informatica, July 1983.

Clark K.L., McCabe F.G., "IC-PROLOG : Language features". Proc. Logic Programming Workshop, Debrecen, July 1980, p. 45-52.

Clark K.L., McKeeman W.M., Sickel S., "Logic program specification of numerical integration". Logic Programming, Academic Press, 1982, p. 173-185.

Colmerauer A., "PROLOG II : Manuel de référence et modèle théorique". GIA, Faculté des Sciences de Luminy, Mars 1982.

Davis R., "Meta-rules : reasoning about control". Artificial Intelligence, vol. 15 n° 3, December 1980, p. 179-222.

Davis R., Buchanan B.G., "Meta-level knowledge : overview and applications". Proc. IJCAI 1977, p. 920-927.

Dincbas M., "The METALOG problem solving system : an informal presentation". Proc. Logic Programming Workshop, Debrecen, July 1980, p. 80-91.

Dincbas M., "A knowledge-based expert system for automatic analysis and synthesis in CAD". Proc. IFIP Congress 1980, Tokyo, October 1980, p. 705-710.

Fahmi A., "Contrôle des systèmes de déduction automatique fondés sur la logique". Thèse de docteur-ingénieur, ENSAE, Novembre 1979.

Gallaire H., Lasserre C., "Controlling knowledge deduction in a declarative approach". Proc. IJCAI 1979, p. 51-56.

Gallaire H., Lasserre C., "Metalevel control for logic programs". Logic Programming, Academic Press, 1982, p. 173-185.

Hayes P., "Computation and deduction". Proc. Math. Foundations of Computer Science Symp., Czech. Acad. Science, 1973.

Kowalski R., "Algorithm = Logic + Control". Comm. ACM, vol. 22 n° 7, 1979, p. 424-436.

Kowalski R., Logic for problem solving. North-Holland, 1979.

Pereira L.M.,"Logic control with logic". Proc. 1st Int. Logic Programming Conference, Marseille, September 1982, p. 9-18.

Porto A., "Epilog : a language for extended programming in logic". Proc. 1st Int. Logic Programming Conference, Marseille, Sept. 1982, p. 31-37.

Robinson J.A., Sibert E.E., "LOGLISP : an alternative to PROLOG". Machine Intelligence 10, 1982, p. 399-419.

Van Emden M.H., "Programming with resolution logic". Machine Intelligence 8, 1977, p. 266-299.

Weyhrauch R., "Prolegomena to a theory of mechanized formal reasoning". Artificial Intelligence, vol. 13, 1980, p. 133-170.