INRIA

UNITÉ DE RECHERCHE
INRIA-ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

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

Rapports de Recherche

N° 763

AN OPERATIONAL FORMAL DEFINITION OF PROLOG
(Comprehensive version of RR 598)

Pierre DERANSART
Gérard FERRAND

DECEMBRE 1987

An Operational Formal Definition of PROLOG

(Revised edition of INRIA report RR 598- October 1987)

Une sémantique Formelle Opérationnelle de PROLOG

(Version révisée du rapport INRIA RR 598 - Octobre 1987)

Pierre DERANSART
INRIA
Domaine de Voluceau
B.P. 105 - Rocquencourt
78153 LE CHESNAY Cedex
☎ : (33-1) 39 63 55 36
uucp: mcvax!minos!ariane!deransar

Gérard FERRAND
Université d'Orléans
Faculté des Sciences-LIFO
B.P. 6759
45067 ORLEANS Cedex 2
☎: (33) 38 41 70 00
uucp: mcvax!inria!univorl!ferrand

Une Sémantique Formelle Opérationnelle de PROLOG

(Version révisée du rapport INRIA RR 598 - Octobre 1987)

Pierre DERANSART
INRIA
Domaine de Voluceau
B.P. 105 - Rocquencourt
78153 LE CHESNAY Cedex
☎ : (33-1) 39 63 55 36

Gérard FERRAND
Université d'Orléans
Faculté des Sciences
LIFO
B.P. 6759
45067 ORLEANS Cedex 2
☎ : (33) 38 41 70 00

Résumé :

Nous présentons dans ce rapport une nouvelle sémantique formelle opérationnelle adaptée à la description des dialectes PROLOG "déterministes" (de type PROLOG Marseille ou Edinbourgh). En particulier, cette sémantique peut être utilisée pour décrire complètement le futur PROLOG standard.

Cette sémantique consiste en un interprèteur abstrait écrit en PROLOG "pur" (avec négation) dont la sémantique est purement déclarative. On peut donc comprendre l'interprèteur sans se référer à aucun dialecte particulier. Bien que fondée sur PROLOG, cette sémantique ne se mord pas la queue. La spécification obtenue n'est pas directement exécutable.

Son avantage principal réside dans le fait qu'elle est compréhensible par toute personne qui connait au moins un dialecte et qu'elle satisfait les critères d'une bonne spécification formelle (abstraction, simplicité, vérifiable, utilisable pour construire des outils de validation, en particulier obtenir une spécification exécutable).

Mots clés :

Programmation en Logique, Spécification Formelle, Sémantique Opérationnelle, Sémantique de PROLOG, Standardisation de PROLOG.

An Operational Formal Definition of PROLOG

(Revised edition of INRIA report RR 598 - October 1987)

Pierre DERANSART
INRIA
Domaine de Voluceau
B.P. 105 - Rocquencourt
78153 LE CHESNAY Cedex
☎ : (33-1) 39 63 55 36

Gérard FERRAND
Université d'Orléans
Faculté des Sciences
LIFO
B.P. 6759
45067 ORLEANS Cedex 2
☎ : (33) 38 63 70 00

Abstract :

We present in that report a *new formal (operational) semantics* well adapted to describe PROLOG's deterministic dialects (Marseille-Edinburgh-like dialects). In particular it could be used in the standardization work of PROLOG.

This semantics is based on an abstract interpreter written in a "pure" PROLOG style (with negation) whose semantics is itself essentially declarative. Thus it can be understood without any reference to any particular dialect, since the negation is treated declaratively (This point will not be developped completely inside this paper). Such a semantics although written in PROLOG, does not depend on itself. It is not an executable specification.

One of its advantages is to be readable without backgrounds by users of a PROLOG dialect. It satisfies also the criteria of a good formal specification of a programming language:

- To be of a very high level, but as simple and understandable as possible.
- To be able to describe a large variety of dialects.
- To describe with the same formalism the whole language.
- To be able to be certified, using validation methods.
- To make easy the production of validation tools, in particular of a runnable specification.

Keywords :

Logic programming, Formal Specifications, Operational Semantics, PROLOG Semantics, PROLOG Standardization.

An Operational Formal Definition of PROLOG

(Revised edition of INRIA report RR 598 - October 1987)

Pierre DERANSART
INRIA
Domaine de Voluceau
B.P. 105 - Rocquencourt
78153 LE CHESNAY Cedex
☎ : (33-1) 39 63 55 36
uucp:
mcvax!minos!ariane!deransar

Gérard FERRAND
Université d'Orléans
Faculté des Sciences-LIFO
B.P. 6759
45067 ORLEANS Cedex 2
☎ : (33) 38 41 70 00
uucp:
mcvax!inria!univorl!ferrand

Note : a revised version of the first edition of RR 598 has been published as [DF 87b]. This is a new augmented edition of RR 598 and [DF 87b] at October 1987.

Cette recherche a été exécutée en partie dans le cadre du GRECO-METHEOL.

1) Introduction:

We present in this paper a *new formal (operational) semantics* well adapted to describe PROLOG's deterministic dialects (Marseille-Edinburgh-like dialects [Col 82, PWB 84]). In particular it is used in the standardization work of PROLOG [DF 86b].
This semantics is based on an abstract interpreter written in a "pure" PROLOG style (with negation) whose semantics is itself essentially declarative. Thus it can be understood without any reference to any particular dialect, since the negation is treated declaratively . Such a semantics although written in a PROLOG syntax, does not depend on itself. Although it is known from the folklore that logic programming can be used as a specification language, it is rarely realized in most of the attempts to use logic programming to specify languages, systems or known PROLOG dialects. Most of the realizations use some unspecified feature (built-in predicate, cut or negation by failure, implicit interpretation strategy), because they are more devoted to simulate an interpreter rather than to specify it. Thus this "PROLOG" semantics is new because of the following points :

- It is completely declarative and formal.
- It has a simple syntax contained in all prolog dialects. However it does not depend on itself. To overcome this difficulty, one uses the notions of relative denotation and declarative semantics of the negation. For these reasons it is not an executable specification.
- It is of higher level than all the others expressed in logic programming style: It uses the search-tree as semantical data.

One of its advantages is that it is readable without major difficulties by users of a PROLOG dialect. It satisfies also the criteria of a good formal specification of a programming language:

- To be of a very high level, but as simple and understandable as possible.
- To be able to describe a large variety of dialects, with the same level of abstraction.
- To describe with the same formalism the whole language.
- To be able to be certified, using validation methods.
- To make easy the production of validation tools of the language implementations.

These points will be discussed in the last section.

This semantics is also interesting because it illustrates the power of logic programming to make specifications. It seems to us that logic programming is generally considered as "impure" executable specification. Our purpose is to show that logic programming may also be used as a perhaps low level but full specification language. Work is actually in progress to apply this idea to the whole project of PROLOG standard [DR 87].
The report is organized as follows: section 2 defines the semantics of PROLOG dialects using the notion of search-tree. Section 3 introduces the formal specification language and its semantics. This section has been expanded in order to give some hints on the declarative negation used in the formal definition language. A new section 4 has been added showing the methodology used to make the specification as clear as possible. The basic aspects of the methodology are related to simple proof schemes of correctness and completness of the specification and introduce the notion of partially formalized comments. In section 5 the data structure representation used in the formal specification is detailed. Section 6 gives a sample of the formal definition for a piece of PROLOG with cut and geler (freeze). A discussion will be presented in section 7.

2) Semantics of PROLOG dialects : the visited-search-tree

Most of the basic concepts are known. They are:

- denotation of a logic program [Fer 85].
- proof tree [Cla 79, DM 85].
- unification [Rob 65].
- search tree [Cla 79, AvE 82, Llo 84].

Their presentation however has been modified in order to make them independent of their historical origin, as proofs by refutation, and to fit exactly with concepts that users of an interactive prolog interpreter may observe (proof tree, answer substitution). Moreover, they permit to simplify the presentation of the logic programming formalism used in the formal definition.

a) Abstract syntax :

This syntax is sufficient to describe the (abstract) syntax of most common PROLOG dialects. It is as follows:

- Terms (untyped). Variable symbols are concretely distinguished from other symbols by beginning with a capital letter. Note that terms may be built with predicate symbols.
- Atoms (atomic formulae) built with predicate symbols and terms. A literal is a positive or negative atom (denoted "not a").
- Clauses $a_o \Leftarrow a_1, \ldots, a_n$ where a_o is the head : an atom, and $a_1, \ldots, a_n$ the body : a list of literals (the order of the litterals in the body is significant). true is the empty body.
- Program : list of clauses (significant order) regrouped into packets.
- Goals are restricted to one literal without loss of generality.

b) Constructive semantics (proof-tree):

The constructive semantics formalizes the notion of proof using the so called proof-trees (also in [Cla 79]). A proof-tree is a finite tree whose nodes are labeled by atoms (or literals in the case of the negation), the root is an instance of a goal, all leaves are labeled by true and all the elementary subtrees are instances of a clause.
Such a tree is the representation of a proof of its root (atomic theorem), using the clause as implications.

Example : P_1 = { plus (o, X, X) ⇐
plus (sX, Y, sZ) ⇐ plus (X, Y, Z) }

```
Proof tree :   plus (so, X, sX)      (proof of ∀ x plus (so, x, sx))
                   |
                 plus (o,X, X)
                   |
                 true
```

The constructive semantics of a program P is the set of all the proof-tree roots. It is declarative and denoted DEN(P), i.e. the denotation of P.

Example :
$\mathrm{DEN}(P_1) = \{\text{plus } (s^n o, t, s^n t) \mid n \geq o, t \text{ term}\}$

c) Logical semantics of a logic program P.

The logical semantics of a logic program P (without negation) is the set of all its atomic logical consequences. This approach is different from the usual one's [AvE 82, Gal 86, Llo 84] which use the Herbrand model. It has been justified in [Fer 85, DF 86a], but some properties of this semantics have been studied in [Cla 79, Fer 85]. They are :

- *It is exactly the set of the proof tree roots, i.e.* DEN(P) (completness results) [Fer 85, DF 86a]
- DEN(P) *is also a model of* P *(also the least in the termal interpretations).*
- DEN(P) *is closed under substitutions.*
- *Given a goal* b *, a PROLOG interpreter builds* answer substitutions σ *such that* $\sigma b \in$ DEN(P).

Example:

Answer substitutions for P_1 and goals :

plus (so, X, Y) ?	σ	$= \{Y \leftarrow sX\}$
plus (X, Y, so) ?	σ_1	$= \{X \leftarrow o, Y \leftarrow so\}$
	σ_2	$= \{X \leftarrow so, Y \leftarrow o\}$

d) Non deterministic operational semantics (search tree) : (procedural semantics in [Cla79])

The answer substitutions are constructed by deriving from a given goal a proof-tree in a top down manner. The construction uses some unification algorithm [Rob 65] and takes advantages of the properties of the unifiers (unique most general unifier- MGU).
We recall here the non deterministic algorithm deriving the proof-trees.

```
b : goal
P : program
T : proof-tree
T : = b ;
while there exists a leaf not labeled by true in the tree T
  do choose l some leaf different from true ;

     choose c: ao ⇐ a1,..., an some clause with variables renamed differently from those
          appearing in T ;

     $\sigma := MGU(l, a_0)$ ;
     $T := \sigma T[\sigma l \leftarrow \begin{array}{c} \sigma a_0 \\ \diagup \quad \diagdown \\ \sigma a_1 \ \cdots \ \sigma a_n \end{array}]$
```

The non deterministic operational semantics is complete. It means that for each σ such that $\sigma b \in$ DEN(P), σb is an instance of the root of some proof-tree obtained by this construction.

Search-tree (also as SLD-tree in [AvE 82, Llo 84])

The non determinism of this algorithm comes from two points :

- non determinism of type l : choice of a leaf ("computation rule" in [Llo 84]). We refer to the l-strategy or l-choice function.
- non determinism of type c : choice of a clause. We refer to the c-strategy or c-choice function.

The "standard l-strategy" consists in fixing the first choice by building the proof-tree choosing the first left-most leaf different from true.

Given a program P, a goal b, and a deterministic l-strategy (choice of the leaves in the proof trees), we can associate to P a unique search-tree in the following way :

- each node is labeled by a list of literals (empty list = true),one of them being distinguished (chosen) by the l-choice function.
- from each node as many arrows are issued as there are heads of clauses in P which can be unified with its chosen atom.
- if $N = b_1, \ldots, \underline{b_i}, \ldots, b_n$ is the node, b_i being the distinguished literal, and there are q clauses in P whose head is unifiable with b_i, presented in the order [1..q], noted $a_j \Leftarrow c$, $1 \leq j \leq q$, then the node N has q sons $N_1, \ldots, N_j, \ldots, N_q$ such that :

N_j = success if N = b, b and a_j unifiable, c = true
$N_j = \sigma(b_1, \ldots, c, \ldots, b_n)$, σ = MGU (b_i, a_j) otherwise.

Now to represent the search-tree we introduce an arbitrary order between the sons which corresponds to the order in which the clauses are organized in P. This representation is called the standard-search-tree (SST).

Example :
Given the program P_2
1 $p(X, Y) \Leftarrow q(X), r(Y)$
2 $p(X, Y) \Leftarrow s(X)$
3 $q(q_1) \Leftarrow$
4 $q(q_2) \Leftarrow$
5 $r(r_1) \Leftarrow$
6 $r(r_2) \Leftarrow$
7 $s(s_1) \Leftarrow$
8 $s(s_2) \Leftarrow$

with the standard l-strategy and the goal p(X, Y), the SST is the following (the chosen atoms are underlined and the substitutions are associated to the arrows) :

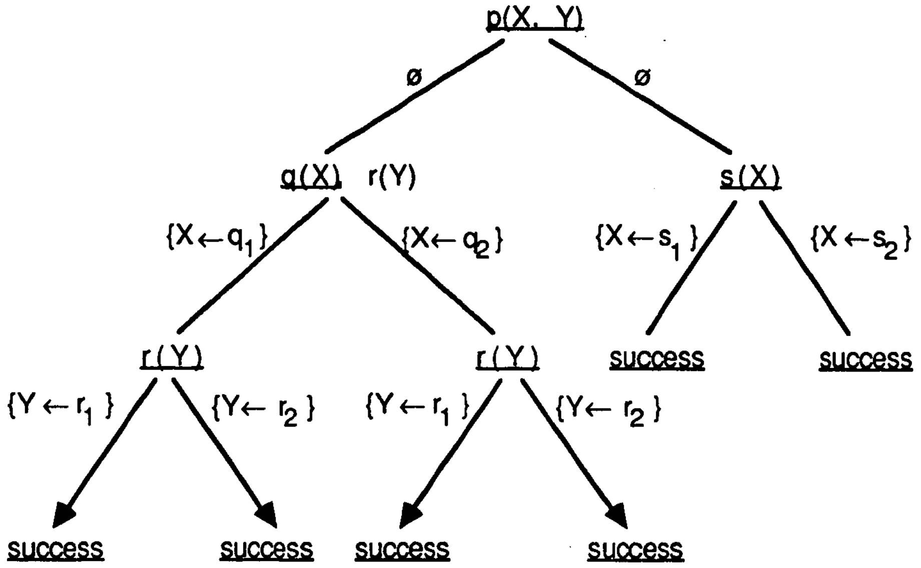


Figure 1

It describes in particular the set of answer substitutions but not the order in which they are obtained. For example the set is the following :

$$\left\{ \begin{array}{ll} X & q_1 , q , q_2 , q_2 , s_1 , s_2 \\ \leftarrow & \\ Y & r_1 , r_2 , r_1 , r_2 , \quad , \end{array} \right\}$$

(The order on the sons is introduced here only for representation purposes).

The search-tree describes all the ways to build all the proof-trees, after giving a l-strategy. A search-tree can be infinite. To any partial finite path issued from the root in the search-tree there corresponds a partial proof-tree. The proof-tree corresponding to a path whose leaf is labeled by success is a complete one.

e) Deterministic operational semantics:

Given a search-tree, fixing the c-strategy consists in giving an (possibly partial) order of tree-walk. The tree-walk defines the order in which the answer substitutions are obtained. The search-tree with its order is called the visited search-tree (VST).
Thus, the operational semantics of a logic program P, given a goal b, is the search-tree derived from the goal with its walk order i.e. the VST. This semantics is sufficient to take into account definite Horn clause programs.

The standard c-strategy consists in a depth-first left to right visit of the standard search-tree (Thus the VST is the SST with the depth-first left to right order on its nodes).

This semantics is particularly well suited to explain the effects of control features like "cut" (dynamic modification of the tree) and delaying primitives (non standard l-choice) [DF 86a,b]. Because the search-tree can have infinite paths, the operational semantics of most of the dialects cannot be described by a single VST, but rather by a (possibly infinite) set of partial VST corresponding to some significant steps of walk. Thus the semantics of a logic program will be the set of partial VST obtained following the "constructive" visit order. Now it should be clear that various semantics could be defined depending on the selected set of partial search-trees representing the semantics.

Remark that this semantics is especially interesting in the following limit cases: if the search-trees are always finite, then the semantics may contain the unique finite VST and it is sufficient to describe the whole semantics of the program (with its goal). On the other hand, if the search-tree is infinite without finite success branches then all partial finite search-trees must be in the semantics.

In the sequel a specific semantics will be chosen.

With the standard strategy the semantics of the program P_1 can be represented by the unique finite VST of figure 1 in which the visit order corresponds to a depth first left to right visit. Now we consider the following modified program :

P_3 : 1 : p (X, Y) ⇐ q (X), !, r (Y)   (introduction of cut)
then the new VST is the same but some branches are not visited and can be omitted as in figure 2.

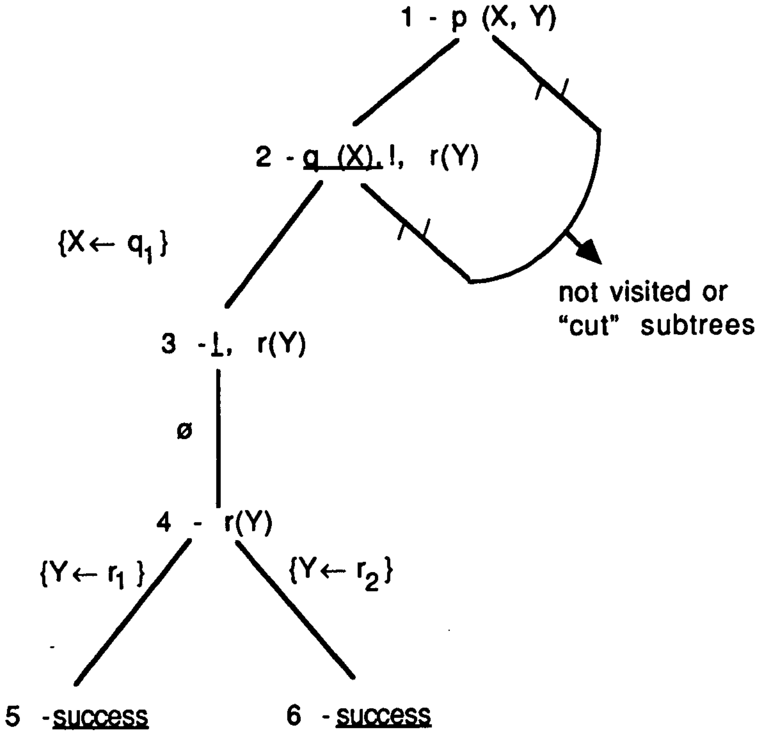


Figure 2

Thus the expected substitutions are :

$$\left\{ \begin{matrix} X \\ Y \end{matrix} \leftarrow \begin{matrix} q_1 , q_1 \\ r_1 , r_2 \end{matrix} \right\}$$

in that order.

The semantics of the cut is clearly defined on the VST by the following : all its branches which are not yet visited and are on the path from the chosen cut to the parent of the node where it appears first remain undeveloped. We illustrate this point by recalling the non-developed branches in the VST by

(Figure 2).

f) The visited-search-tree as semantical objet :

It should be clear that the visited-search-tree is well-suited to describe the semantics of a prolog dialect. First of all it is a mathematical objet, representing all the computations, which is associated to a program of a dialect and a goal. In that sense it can be viewed as the denotation of a program and a goal.

From a practical point of view it is a relatively wellknown object since most of the non standard strategies, and especially c-strategies, are well illustrated on the search-tree ("vertical" or "horizontal" strategies for example, infinite branches eliminations, negation as failure ...).

However one can object that it is not a very abstract object since it uses the same syntax as the analysed program and it cannot be practically obtained without "running" the program. Consequently few properties can be formally proven with such an object.

Our feeling is different. The properties we want to establish are relatively simple : in general by considering non-standard dialects we want to take into account operational properties and thus in most cases if two search-trees are different, then the semantics of the corresponding programs with goals are effectively different. In other terms two programs (with goals) are equivalent if and only if they have the same visited search-tree. Thus we can expect that a clear description of the effect of every primitive of a dialect on the visited-search-tree will be sufficient to understand the semantics of a program with its goal.

It is also clear that this kind of program equivalence is too strong. In some cases we are interested only in considering that two logic programs are equivalent if and only if they produce the same set of answer substitutions in the same order.

At the moment no formal proof methods have been developed using the visited search-tree. However the known properties of the search-tree come as a positive and hopeful argument since the set of the computed answer substitutions does not depend on the l-strategy (switching lemma in [Llo 84], [DF 87]) and thus on the search-tree representation. Thus it appears that a lot of useful properties concerning the computed answer substitutions will be relatively easy to establish by considering the search-trees.

We illustrate and achieve this presentation by giving two examples :

Examples :
Consider now the program P_1 with a non-standard control primitive : a "geler" (freeze) primitive :

P_4 : 1 : p(X, Y) ⇐ geler(X, q(X)), r(Y)

The expected semantics of "geler" is the following : this atom is selected if the variable in its first argument is instantiated, the standard l-strategy is used otherwise.

Thus the semantics of "geler" can be well described by fixing the l-choice function. This gives the VST of figure 3.

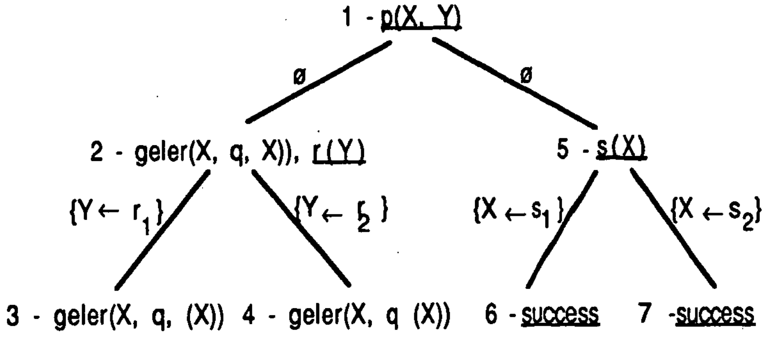


Figure 3

Nodes 3 and 4 correspond to failure node as no literal can be chosen, since "geler" has not been unfrozen. It is easy to see also that with the clause :

1' : p(X, Y) ⇐ geler(X, q(X)), r(Y), !

no answer substitution will be obtained.

Another application is to consider the semantics of the disjunction as (; is treated as a predicate)

(A ; B) ⇐ A
(A ; B) ⇐ B

Thus the program P_3 can be written as :

P_5: 1. p(X, Y) ⇐ ((q(X), !, r(Y)) ; s(X))
1'. p'(X, Y) ⇐
2. q(X) ⇐ ($X = q_1$; $X = q_2$)
3. r(X) ⇐ ($X = r_1$; $X = r_2$)
4. s(X) ⇐ ($X = s_1$; $X = s_2$)
5. X = X ⇐

which is equivalent to P_3 in the sense of the list of the answer-substitution. In fact it is easy to depict its VST (figure 4).

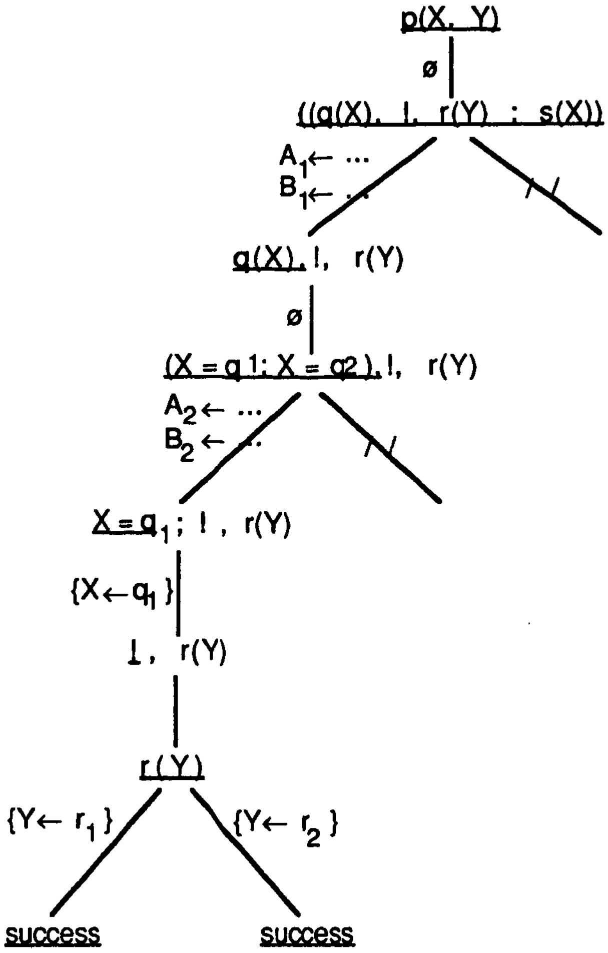


Figure 4

In this semantics the cut “goes through” the disjunction, i.e. it “cuts” the choices of the parent of the body in which it appears. The program produces the same answer substitutions because the “predicate variables” A_i and B_i do not mixt with “term variables” X and Y.

3) **Semantics of the formal specification language** :

The formal specification language is a very simple logic programming dialect which we describe now together with its semantics.

a) Abstract syntax :

As in section 2-a, but the order of the clauses in the packets as well as the order of the literals in the body of the clauses is irrelevant. Function and predicate symbols are disjoint sets.

b) Relative denotation :

The notion of denotation is sufficient to describe the semantics, but to deal with the PROLOG dialects which contain built-in predicates, an extended notion of denotation is needed: the relative denotation [Fer 85].

Given a set E of atoms, we denote E the set of clauses $\{A \Leftarrow \mid A \in E\}$. The denotation of the logic program P relative to E, denoted DEN(P|E) is the denotation DEN(P $\cup$ E) of the logic program P $\cup$ E.

For example consider a predicate ℕ-**plus**(X, Y, Z) which satisfies Z = X+Y (on natural integers

$\mathbb{N}$). E contains all the clauses like :

$\mathbb{N}$-**plus**(X, Y, Z) ⇐

for all X, Y, Z integers such that Z = X+Y.

The logical semantics of a logic program P using this predicate (not defined in P) is exactly

$$DEN(P|E) = DEN(P \cup E).$$

The relative denotation is reserved to very simple and known domains like integers, reals, lists for which a formal definition in Horn clause style would be unnecessarily complicated.

c) Proof-trees with negation

In the formal definition we want to avoid the use of control features -like the cut- which can be only understood by reference to some interpretation strategy. But in order to preserve a good expressiveness of the formal specification language we will make use of the negation which may be nicely understood declaratively (consider for example what should be a specification of "not integer"). Various logical semantics can be given to logic programs with negation [Llo 84, Del 86]. No one is satisfactory from the declarative point of view, especially for logic programs with negation.

We have chosen a notion of negation which has a clear declarative semantics. This semantics is based on a generalization of the notion of proof tree (the proof-tree is a declarative notion, unlike the construction of a proof tree). In [DF 87a], [Fer 86] declarative and operational semantics of Logic Programming with such a negation are studied, and the links between this kind of negation and the usual implementation of the negation in Prolog are completely developed. (This negation is also compared with the theory based on the completion of a program [Llo 84]). In particular this negation is compatible with what is considered as a "safe use" of the negation in Prolog [Llo 84]). This shows that the negation used inside the specification will permit to transform it into a runnable specification without major difficulties.

Now we describe the main lines of the foundations of this negation.

The idea is the following :

In a proof tree, a leaf may be a negative literal "not a" iff no instance of "a" is the root of a proof-tree.

(In a proof tree we can see clauses as inference rules. So remark the similarity of this idea with the ideas of Closed World Assumption and Negation as failure).

In general this first idea is inconsistent (take the clause p ⇐ not p, so p is a proof tree root iff p is not a proof tree root ! But we consider only a class of programs, for which such a notion of proof tree with negation is founded. It is a generalization of the stratified programs of [ABW 86].

This class, the "finitely founded programs", seems large enough for Logic Programming practice (discussion of this point in [DF 87a]).

Intuitively, in a finitely founded program, a predicate cannot be defined by its own negation. The idea is to rule out "vicious circle" by dividing the atoms into levels. The precise definition is in [DF 87a]. For example, let us consider

a ⇐ b, not c

which is an instance of a clause of the program.

Then we have

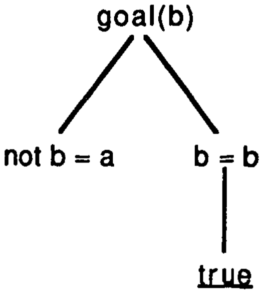
$$\text{level}(b) \leq \text{level}(a)$$
$$\text{level}(c) < \text{level}(a)$$

We denote by NEG(P) the set of all the negative literal "not a" such that no instance of "a" is the root of a proof-tree. So in a proof-tree, the negative leaves which are allowed are the elements of NEG(P).

Example : the following program P is finitely founded.

```
X = X   ⇐
goal(X) ⇐ not X = a, X = b
```

not(b = a) is in NEG(P)
and here is a proof-tree :



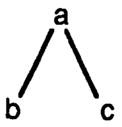
Remark : if we quit our declarative framework, it is well known that, with the goal "goal(X)" and a standard interpreter, we have a failure. However this does not prevent the proof-tree from existing ! It is only an illustration of an "unsafe use" of the negation of Prolog ([Llo 84]).

d) Declarative semantics of logic programs with negation

The constructive semantics of a program P is the set of all the proof-tree roots with negation : we call it the denotation of P : DEN(P). It is a declarative semantics, which is a generalization of the case where P is without negation. Note that a negative literal "not a" is in NEG(P) iff no instance of "a" is in DEN(P).
From a logical point of view, when P is without negation, we have seen in 2)b) that DEN(P) plays two roles at once : it is a set of logical consequences and it is an interpretation. Many important properties of this logical semantics can be extended to the general case (P finitely founded). However the set of axioms which defines this logical semantics is not P itself anymore. In fact, there are axioms which depend on NEG, but this logical semantics cannot always be defined by a recursively enumerable set of axioms. These problems are studied in [DF 87a].

So there is a logical semantics which is equivalent to the constructive semantics, but the more important aspect is the constructive one. It gives to the negation a declarative meaning which is clear and easy to use. Let us illustrate this with examples.

Examples : we describe the graph



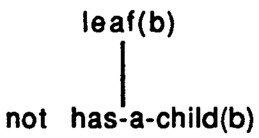
by the clauses

```
          parent(a,  b)      ⇐
          parent(a,  c)      ⇐
```

and we add :

```
          has-a-child(X)     ⇐     parent(X,  Y)
          leaf(X)            ⇐     not  has-a-child(X)
```

Let P_1 be this new program. not(has-a-child(b)) is in NEG(P_1), and here is a proof-tree :

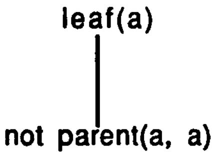


So leaf(b) is in DEN(P_1).

Now, let P_2 be the program.

```
parent(a, b)    ⇐
parent(a, c)    ⇐
leaf(X)         ⇐    not parent(X, Y)
```

not(parent(a, a)) is in NEG(P_2),and here is a proof-tree :



So leaf(a) is in DEN(P_2) !

Hence this last program P_2 is not correct (in an obvious sense, w.r.t. the graph).

These two examples show how the proof-tree with negation is a useful tool to understand the declarative meaning of the negation.

Remark : if we quit our declarative framework, a standard interpreter does not compute this last tree. It is because the interpreter makes here an "unsafe use" of the negation of Prolog (variable Y). Our notion of negation permits the distinction between declarative and operational points of views, and in particular it clarifies correctness questions.

4) Formal specification design :

a) Principles:

The design of a good formal specification seems in general to be a kind of art. It seems that nothing can be formalized to describe what can be denoted as the clarity of the specification. Nevertheless we can point out the following qualities:

α) The specification should not describe what should not be specified. For example if we do not want to specify the representation of the substitutions, we try to design the specification without giving any representation of it. In fact it would be difficult, if doing so, to ensure that no specific property of the substitutions would be implicitly included in the specification and thus could be used later on by some implementer as part of the specification. One way to avoid such drawbacks is to avoid as much as possible to represent what is not needed to understand the semantics. One advantage is the reduction of the size of the specification by decreasing the number of used concepts and data structures.

In our specification this reduction is obtained also by using the notion of relative denotation, which avoids to give an explicit representation of some operations like unification, variable renaming, arithmetic operations ...

On the other hand this increases the gap between the formal specification and the runnable specification. But it should be clear that a runnable specification, even if completely formal, will contain a number of unnecessary details as mentionned above since some choices of representation have to be included.

β) The specification should be easy to read. It comes as an evidence that giving the specification by clauses (even limited to Horn clauses with "simple negation") is not sufficient to guarantee the simplicity of the specification. Something more has to be clearly understandable by the reader. We can qualify it by the properties the written axioms have to satisfy. The questions are: on one hand, are the axioms correct in the sense that what they specify are desired transformations of the search-tree. This can be denoted as the correctness of the specification. On the other hand, do we write enough axioms to specify the semantics of all the desired programs ? This can be denoted as the completeness of the specification.

It seems to us that if the reader is in the best conditions to understand that the specification is correct (this means that he understands which are the transformations) and that it is complete (he understands that the transformations are completely defined), he will get a good understanding of the transformations and thus of the whole specification.

Our conclusion is the following: the design of the specification is basically guided by the readability of the proofs of correctness and completeness of the specification. It comes as an evidence that easier to imagine is the proof, simpler to understand is the specification.

Now the design of the specification will be organized such that the proofs are easy to perform: we will see that the order of the literals in the body of the clauses follows the order in which the correctness proof is performed, and the order of presentation of the clauses follows the order in which the completeness proof is easier to perform.

Moreover, to help the reader, we give him the assertions (or in order not to overload the specification an abstract of them) used to perform the proofs. Thus the specification is no more limited to the clauses but to the clauses plus the assertions.

It has to be clear that the assertions are not part of the formal specification, since they are not completely formalized. They are used only as comments to help the reader to understand the axioms. But these assertions are written following a rigorous methodology which consists in using these assertions to perform the correctness and completeness proofs.

One aspect of the design of the specification is to restrict the kind of assertion to relatively simple ones and the kind of proof using a very simple proof scheme. We can say that sometimes the simplicity of the proof will be viewed as more important than the simplicity of the axioms, if this seems to improve the readability of the specification. Sometimes the axioms will be written with redundancies using negation to point out clearly the considered cases. This way appears as a heavy style from the "executable specification" point of view. But it helps indeed to improve the readability of the specification by facilitating the completeness proof.

It must be noted that this attitude is permitted by the language of formal specification in which such a methodology can be developed. We present here briefly this methodology.

b) Methodology:

α) The axioms:

The axioms have to be easily understandable. With axioms without negative literals there is no difficulty: they are universally quantified definite Horn clauses and they have a logical and equivalently constructive meaning as recalled in section 2.b.

We need to consider more carefully the intended meaning of the axioms with negation. As noticed in section 3.c, the axiom

$$\text{not parent}(X,Y) \quad \Rightarrow \quad \text{leaf}(X) \qquad (1)$$

should be considered with universal quantification:

$$\forall X,Y(\text{not parent}(X,Y) \quad \Rightarrow \quad \text{leaf}(X))$$

or

$$\forall X((\exists Y \text{ not parent}(X,Y)) \Rightarrow \quad \text{leaf}(X)) \qquad (1')$$

This is clearly not the desired meaning. We wanted to axiomatize

$$\forall X((\forall Y \text{ not parent}(X,Y)) \Rightarrow \quad \text{leaf}(X)) \qquad (2)$$

(or $\forall X((\text{not } \exists Y \text{ parent}(X,Y)) \Rightarrow \text{leaf}(X))$
or X is a leaf if X has no son).

Thus, as shown in section 3.d, in order to formalize this idea with universally quantified clauses, we need to write:

$$\forall X(\text{not has\_a\_child}(X) \quad \Rightarrow \quad \text{leaf}(X)) \qquad (3)$$

$$\forall X,Y(\text{parent}(X,Y) \quad \Rightarrow \quad \text{has\_a\_child}(X))$$

Hence there is no ambiguity now.
As it comes also as an evidence that it could be easy to write down axiom (1) when thinking to axiom (2), we need a simple rule to enforce to write (3) if the meaning is (2) and (1) if the meaning is (1').

On one side we can observe that if we consider instances of the axioms in which the negative literals are ground there is no ambiguity since the lack of variables renders quantification meaningless (not ∃..parent(...) or not ∀..parent(...)).

On the other side we observe also that we are not interested in every proof-tree (of the specification), since the root of every meaningful proof-tree will have the form:

semantics(P,G,T)

where P and G are ground terms (the abstract terms for a program and a goal are ground terms from the specification point of view).

Hence if we are able to prove that every proof-tree whose root satisfies P,G ground can be built with instances of clauses in which the negative literals are ground, we are in condition to read the axioms easily, since no confusion about quantification of negative literal may arise.

From the reader point of view all things are simplified if the proof of negative literals closure is easy to perform and if this fact comes as an evidence consequently. The correctness proof method will include such a proof.

β) Comments:

The comments are assertions written in natural language style attached to every predicate. For example to '**semantics**' it could be attached the following "assertion":

semantics(P,G,T) then if P is a program and G a goal then T is a tree representing the partial search-tree obtained after a finite number of constructive steps starting with P,G.

The following points should be noted:

- The first part (before the first 'then') have to be read: 'if **semantics**(P,G,T) is a proof-tree root'. This justifies the 'then' and shows that the assertion is a correctness assertion, i.e. satisfied by every proof-tree root (in the formal specification language).

- The assertion has the form:

$$A(P,G) \Rightarrow B(P,G,T)$$

where A(P,G) describes the type of the arguments and contains implicitly a closure condition of P and G (since P,G are interpreted as abstract terms of the described dialect).

- The completeness of the semantics is guaranteed if it is possible to show that for every P program, G goal and every partial search-tree T which corresponds to a finite number of constructive steps starting with P,G, there exists at least a proof-tree (built with clauses of the specification) whose root is **semantics**(P,G,T). Thus in the assertion there is also a completeness assertion: it is A(P,G) and B(P,G,T) meaning that if this condition is satisfied we can ensure the existence of a proof-tree whose root is **semantics**(P,G,T).

Now we give the general form of the comments.

Let p be a predicate.

We say that p is correct w.r.t. the assertion S if, in all the atoms p(...) of DEN (i.e. p(...) is a proof-tree root), the arguments verify S.

We say that p is complete w.r.t. the assertion C if all the atoms p(...) whose arguments verify C are in DEN.

Here we are only concerned about assertions with a particular form:
(x,y denote a partition of the arguments of p)

p correct w.r.t.

(S) if ground(x) and A(x) then ground(y) and B(x,y)

p complete w.r.t.

(C) ground(x) and ground(y) and A(x) and B(x,y)

This is expressed by the designer by the following information specified in the comments which by the way privileges the correction aspect, but permits to distinguish which variables are supposed to be ground, which of them are not

p(x,y) then if A(x) then B(x,y).

(if there is no A it is thus possible to write the comments : p(x,y) iff B(x,y)).

These assertions are proven using a method which is simple enough to be intuitively followed. It is out of the scope of this presentation to justify the method. We will only show in an example how a proof can be performed and thus how a reader can get some confidence about the correctness and completeness of axioms.

However, because of the negation it is useful to mention another point: we say that

not p is correct w.r.t. S' if, in all the literals not p(...) of NEG, the arguments verify S'.

not p is complete w.r.t. C' if all the literals not p(...) whose arguments verify C' are in NEG.

Now, we have the following properties: (read (S) and (C) as above)

If p is correct w.r.t. (S) then not p is complete w.r.t.:
ground(x) and ground(y) and A(x) and not B(x,y)

If p is complete w.r.t. (C) then not p is correct w.r.t.:
if (ground(x) and ground(y) and A(x)) then not B(x,y)

Correctness (completeness) proofs use correctness (completeness) of some not p, and so use completenes (correctness) results. There is no contradiction because of the stratification of the program (section 3.c). In order to use correctness of not p, the proof needs to verify the condition ground(x) and ground(y) and A(x). In this verification, ground(x) and ground(y) correspond to the fact that in proof-trees negative leaves are ground.

Sometimes in clauses with a negative literal the valid assertion not B is not the most convenient one to follow up the proof of correctness. It is possible to help the reader by giving an assertion not B' such that B' is stronger than B and which is not necessarily easy to find. Thus it could be formulated as the comment : not p(x, y) then not B'.

γ) The proof method:

The proof method consists in three steps:
- correctness, proven in every clause
- completeness, proven in every packet and clause
- stratification, proven on the whole specification.

One element of the simplicity of the method is its modularity. Main properties are proven inside the clauses, or considering the heads of a packet without looking at other parts of the program. This is possible because of the given comments which contain all the needed assertions to perform the proofs locally. This is exactly what we want: to be able to get the results by looking only at the clauses or at most the packets. The stratification can be automatically verified and is guaranteed by the designer.

Correctness proof:

In every clause $p_0() \Leftarrow q_1(),\dots,q_n()$ ($q_k = p_k$ or $q_k =$ not p_k)
let if P_0 then Q_0 be the correctness assertion attached to p_0 (denoted by S above) and let if P_k then Q_k be the correctness assertions attached to the q_k
(if q_k is positive, P_k = ground(x_k) and A(x_k), Q_k = ground(y_k) and B(x_k,y_k),
if q_k is negative, P_k = ground(x_k) and ground(y_k) and A(x_k), Q_k = not B(x_k,y_k))
The proof follows the scheme given in figure 5, which can be interpreted: the head of the clause is supposed to satisfy the hypothesis P_0 , then the proof is performed in every literal following the left to right order (denoted by k ,$1 \le k \le n$). Every hypothesis of the correctness assertion attached to the literal $q_k()$ is proven using all results obtained before (for q_j, $j<k$). Then Q_0 is proven using all properties assumed in the body and P_0.

This strict order makes the proof simple to follow and consistent. It guarantees that every proof tree root will satisfy the correctness assertion (and the "simplicity of the negation", both are proven in the same step).

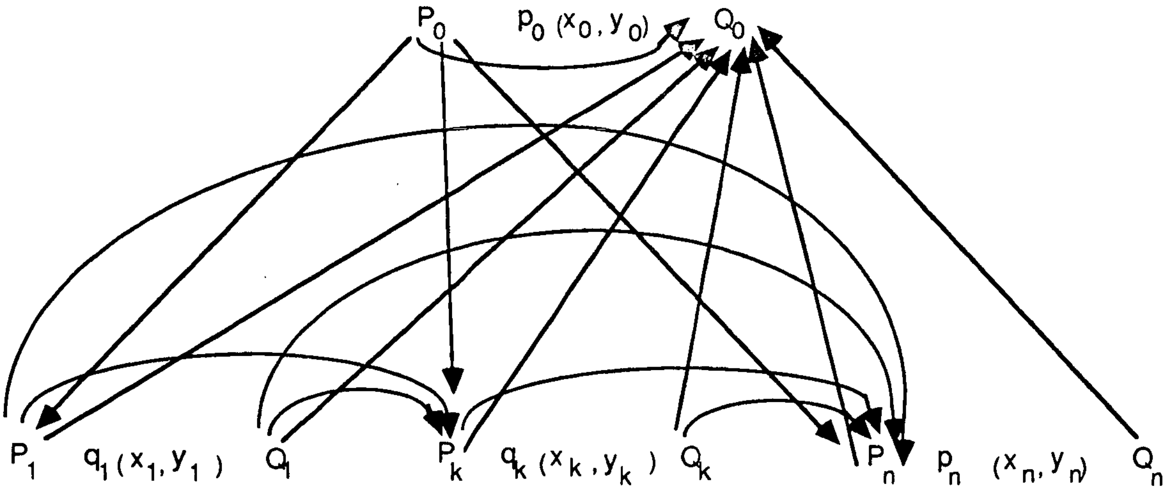


Figure 5

Completeness proof:

We give only the most intuitive elements of the proofs. It consists in proving that for every atom p(x,y) which satisfies C= ground(x) and ground(y) and A(x) and B(x,y) there exists at least one proof-tree. The idea of the proof is to show that all possible (generally exclusive) cases are considered in the packet of 'p', hence at least one clause may be used to develop a proof-tree from the considered goal. It remains to show that the completeness properties are preserved inside each clause of the packet and that the construction will terminate (necessarily successfully) by exhibiting some decreasing criterium. Thus the proof is performed in three steps:

1- The completeness conditions C are split into m conditions C_j ($1 \leq j \leq m$) if there are m clauses in the packet of 'p'. Thus one have to show that C implies the disjunction
C_1 or C_2 or ... or C_m, where the C_j assertions are such that if C_j is satisfied by terms x, then x is an instance of the head of the j^{th} clause.

2- In every clause the proof follows the scheme given in figure 6, which can be interpreted often as: if the head is ground and satisfies its attached condition $C_{0,j}$ then there exists a ground instance of the body in which all literals $q_k()$ satisfy their condition C_k.
(if q_k is positive, C_k = ground(x_k) and ground(y_k) and A(x_k) and B(x_k,y_k),
if q_k is negative, C_k = ground(x_k) and ground(y_k) and A(x_k) and not B(x_k,y_k)).

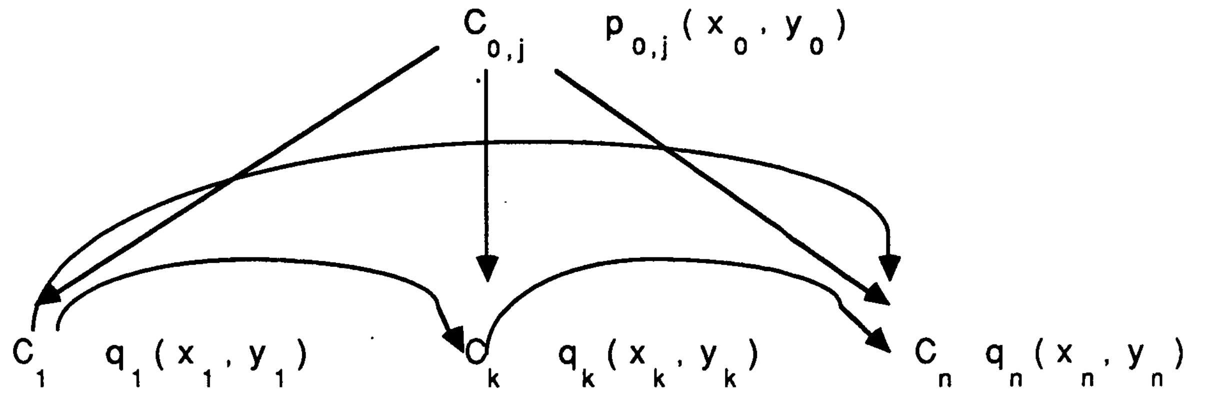


Figure 6

3- The fact that something decreases and thus the construction always terminates successfully will be guaranteed by the designer.

As a conclusion it should be clear that we do not claim that the reader should be able to perform such proofs to understand the specification. The designer is expected to guarantee that such proofs have been performed following the given scheme. This comes only as an element of reliability of the specification and also, as argued at the beginning, of its clarity.

δ) Example:

In order to illustrate briefly the methodology, we give all the correctness and completeness proofs of a very short program with negations.

includ(L_1,L_2) then if L_1 and L_2 are ground lists then $L_1 \subset L_2$

includ(L_1,L_2) ⇐ not **nonclud**(L_1,L_2)

nonclud(L_1,L_2) then if L_1 and L_2 are ground lists then $L_1 \not\subset L_2$

nonclud(L_1,L_2) ⇐ **elem**(A,L_1), not **elem**(A,L_2)

elem(A,L) then if L is a ground list then A is ground and A ∈ L

```
elem(A,[A|L]) ⇐
elem(A,[B|L]) ⇐ elem(A,L)
```

Correctness proofs:

elem (1^{st} clause): $[A|L]$ is supposed to be a ground list then A is ground and $A \in [A|L]$

elem (2^{nd} clause):

[B|L] is supposed to be a ground list.

We can use (body of the clause): if L is a ground list then A is ground and $A \in L$.

Now L is a ground list (because of [B|L]) so A is ground, and $A \in [B|L]$ (because $A \in L$).

ninclud:

L_1 and L_2 are supposed to be ground lists.

We can use (1^{st} literal in the body): if L_1 is a ground list then A is ground and $A \in L_1$.

Now L_1 is a ground list, so A is ground and $A \in L_1$.

We can use (2^{nd} literal in the body): if A and L_2 are ground and L_2 is a list then not $(A \in L_2)$

Now A and L_2 are ground and L_2 is a list so $A \notin L_2$.

It follows that $L_1 \not\subset L_2$.

includ:

L_1 and L_2 are supposed to be ground lists.

We can use (body):

if L_1 and L_2 are ground lists then not $(L_1 \not\subset L_2)$.

Now L_1 and L_2 are ground lists, so $L_1 \subset L_2$.

Completeness proofs:

The completeness conditions C are:

for **elem** : L is a ground list and A is ground and $A \in L$.

for **ninclud** : L_1 and L_2 are ground lists and $L_1 \not\subset L_2$.

for **includ** : L_1 and L_2 are ground lists and $L_1 \subset L_2$.

elem : C can be splitted in C_1 or C_2, where

C_1 is: L is a ground list and A is ground and A is the head of L

C_2 is: L is a ground list and A is ground and A is in the tail of L.

In both cases if C_1 (C_2) is satisfied by element and list terms, these are instances of the corresponding heads..

In each case the body satisfies the completeness condition.

The decreasing criterium is the length of the second argument (which is a ground list).

ninclud : As the head of the clause has two different variables only the first condition is trivially satisfied. In the body the completeness conditions for the two literals are respectiveley:

L_1 is a ground list and A is ground and $A \in L_1$.

L_2 is a ground list and A is ground and not $(A \in L_2)$.

Now by hypothesis $L_1 \not\subset L_2$, so there is an A in L_1 such that $A \notin L_2$,, so there is an instance of the body where each literal satisfies the completeness condition.

includ : as the head of the clause has two different variables only the first condition is trivially satisfied. In the body the completeness condition for the literal is:

L_1 and L_2 are ground lists and not $(L_1 \not\subset L_2)$.

Now by hypothesis $L_1 \subset L_2$, so it is obvious that in the body the literal satisfies the completeness condition.

Stratification: The levels of **elem, ninclud, includ** are respectively 0,1,2.

It is easy to see , in this very short program, why the proof method works:

correctness (completeness) of **includ** comes form
correctness (completeness) of not **ninclud**, which comes from
completeness (correctness) of **ninclud**, which comes from
completeness (correctness) of **elem** and not **elem.**

completeness (correctness) of not **elem** comes from
correctness (completeness) of **elem**, which comes from inductive reasonning without problem of negation.

5) Basic elements of the formal definition :

The formal definition will be given by a logic program (definite Horn clauses program with negation) describing the search-tree associated to some given program P together with a goal. Thus the operational formal semantics of a program P and a goal g is the declarative semantics of this program.

More precisely it corresponds to the relative denotation of the relation :

semantics (P, G, T)

whose assertion is:

semantics(P,G,T) then if P is a program and G is a goal then T is a partial search-tree issued from G which is complete (if T is finite) or limited to some last constructed finite success branch following the tree-walk order of T

(Thus only programs and goal with at least one answer substitution obtained following the program strategy or with finitely failed search-tree will be considered in this sample of formal definition).

The denotation of **semantics** contains all partial search-trees corresponding to a success branch obtained by the specific strategy of P and the complete one if there exists a finite search-tree. For simplicity we have restricted the semantics considered only to steps corresponding to success paths or complete finite search-trees. This kind of limitation could be easily suppressed to take into account infinite computations without any success path.

It is worth noticing that the intended semantics is very simple : the (non)standard strategies of P can be described using only a logic program without references to any strategy.

To understand the formal semantics one must know the used data structures : relative denotation, programs and search-tree.

L-language

It is supposed that predicates and functions of the described language are written in a defined language denoted L. It contains all possible symbols, including variables names which are considered as constants in the formal semantics. In the following it will be necessary to rename variables.

We will use, as in [JM 84] integers to rename denumerable sets of variables. The used integers will be represented in lists denoting in a Dewey like notation the nodes of the search-tree also.

Relative denotation

As it has been shown, the relative denotation is a way to give a meaning to the following predicates which will be used in the formal definition without logic description.

L-var(X)	iff X is a variable in L.
L-term(X)	iff X is a free term in L.
L-ground(X)	iff X is a ground term in L.
L-instance(T_1, S, T_2)	iff T_2 is an instance of T_1 by the substitution S.

L-unify (T_1, T_2, S) iff S exists and S is the MGU of T_1 and T_2.

L-rename(I, T_1, T_2) iff T_2 is T_1 with renamed variables using the integer (or indice) I.

Notice that we could define :

L-unifiable$(T_1, T_2) \Leftarrow$ **L-unify**(T_1, T_2, S).

We denote by = the L-syntactic equality
($T_1 = T_2$ iff T_1 and T_2 are the same L-terms).

Structure of a program P

P is a list of clauses. The clauses defining a predicate are grouped in a list called **packet**. This later predicate will not be described here. Its meaning is the following :

packet(P, I, A, Q) then if P is a program, I an integer used to rename the variables of L, A an atom then Q is an instance of the packet associated with A in P, or nil if there is no packet.

Structure of the search trees :

A search-tree is :
a node : node (I, G)
I list of integers built with o and succ (successor).
G goal = list of literals.
(lists are dotted. Empty list is nil.)
or mk-tree (N, F)
N node
F list of search-trees.

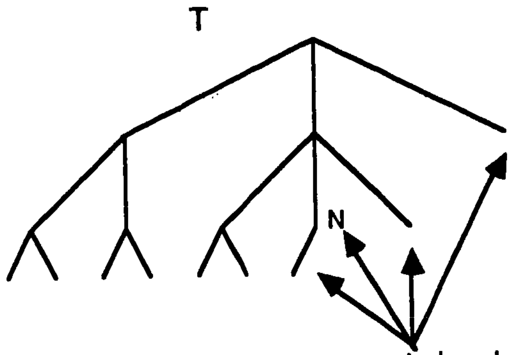
Note that all the nodes in a search-tree are different.

A goal is represented by a list of literals denoted differently:

true	(empty goal)
A + L	literal A, L goal.

Search-trees are constructed by extensions or modified (the cut modifies and extend a search-tree).

In the whole program a tree argument is denoted by a term T-N where T is a tree (or a list of trees) and N a node in T, which is in general the "current node in T", i.e. the last visited node in the tree-walk order of the SST. Note that T-N is a partial SST such that all the nodes following N are hanging nodes as illustrated in figure 7. This notation avoids the use of two arguments where it is unneccessary. Anonymous variables, when they are not necessary to understand the definition, are written _ as in [PWB 84].



not already treated nodes (hanging nodes)

Figure 7

The program uses tree-walk functions defined as follows :

first-son(T-N_1, N_2) then if N_1 is a node of the search-tree T then N_2 is the first son of N_1.

son(T - N_1, N_2) then if N_1 is a node of the search-tree T then N_2 is one of the sons of N_1.

brother(T-N_1,N_2) then if N_1 is a node of the search-tree T then N_2 is the first brother of N_1.

father(T - N_1, N_2) then if N_1 is a node of the search-tree T then N_2 is the father of N_1.

N in T iff N is a node of T.

if G is a goal and A an atom then:

tail(G_1, G_2) iff G_2 is the list rest of G_1.
head(G, A) iff A is the head atom of G.

Definitions are obvious and will not be recalled here [DF 86b]

has-a-son(T-N) then if N is a node of the search-tree T then N has a son in T
has-a-brother(T-N) then if N is a node of the search-tree T then N has a brother in T
has-a-father(T-N) then if N is a node of the search-tree T then N has a father in T

6) A sample of formal specification:

```
semantics(P, G,T)⇐
  buildtree(P,mktree(node(o.nil,root), node(o.nil,G).nil) -node(o.nil,G), T) .
```

buildtree(P,T_1-N_1,T_2-N_2) then if P is a logic program and T1 is a partial search-tree whose current node is N1, then T_2 is the partial search-tree obtained by a constructive tree-walk of T1 from N1 until a success node has been reached (N2 = node (I, true)), or a complete finite search-tree has been obtained (N2 = node (o.nil, root)).

```
buildtree(P,  T-N,  T-N)  ⇐  N = node(l, true)

buildtree(P,  T-N,  T-N)  ⇐  N = node(o.nil, root)

buildtree(P,T1-N1,T2)     ⇐  not N1=node(o.nil,root),
                             treatment(P,T1-N1,T3-N3),
                             c-choice(T3-N3,N4),
                             buidtree(P,T3-N4,T2)
```

Three not exclusive cases are considered here: T is the desired semantics (success branch or completed tree) or T_1 and T_2 are different and at least one treatment is necessary to obtain T_2.

Only a very restricted part of a possible specification is given herein. The treatments are thus very simple ones. Morever, we will suppose that the intended c-strategy of a program corresponds to a top-down left to right standard search-tree-walk.

c-choice(T-N_1,N_2) then if N_1 is the current node of the search treeT then N_2 is the next leaf node following the standard search tree walk order, if any, or the root, if no one exists.

```
c-choice(T, N)            ⇐  first-son(T, N)

c-choice(T, N)            ⇐  not has-a-son(T), brother(T, N)

c-choice(T,  N)           ⇐  not has-a-son(T),
                             not has-a-brother(T),
                             re-c-choice(T,  N)


re-c-choice(T,  N)        ⇐  brother(T,  N)

re-c-choice(T-N,  N)      ⇐  not has-a-brother(T-N),
                             not has-a-father(T-N )


re-c-choice(T-N1,  N2)    ⇐  not has-a-brother(T-N1),
                             father(T-N1,  N3),
                             re-c-choice(T-N3, N2)


has-a-son(T)              ⇐  son(T,_)
has-a-brother(T)          ⇐  brother(T,_)
has-a-father(T)           ⇐  father(T,_)
```

treatment$(P,T_1\text{-}N_1,T_2\text{-}N_2)$ then if P is a program, T_1 a search tree, N_1 the current node of T_1 then T_2 is T_1 where N_1 has been expanded if the chosen goal (if any) has clauses whose heads can be unified, or T_1 modified if the chosen goal is a cut. In all cases the current node is unchanged, i.e. N_2=N_1.

```
treatment(P, T-N,  T-N)          ⇐ N = node(_, true)
treatment(P, T-N,  T -N)         ⇐ N = node(o.nil,  root)

treatment(P, T-N, T-N)     .     ⇐ N = node(I, G),
                                   not G = root,
                                   not I-choice-succeeds(G)

treatment(P,  T1 -N,  T2)        ⇐ N = node(I,  G),
                                   I-choice (G, !(J)) ,
                                   treat-cut(T1-N, T2,  J)

treatment(P,  T1-N,  T2)         ⇐ N = node(I, G) ,
                                   I-choice(G, A),
                                   not is-a-cut (A),
                                   expand(P,  T1-N, A, T2)
```

Note that the predicate cut ('!') is flagged in order to be able to detect the nodes in which the choices have to be cut. The cuts are flagged by **expand**.

```
I-choice-succeeds(G)             ⇐ I-choice(G, A)
is-a-cut(A)                      ⇐ A = !(J).
```

```
I-choice(G, A)                   ⇐ head(G, A)
```

This is the standard choice function of type I (first literal in the goal list). To take into account a "geler" (freeze) predicate could give the following description :

```
I-choice(A+L, B)        ⇐ not is-geler-atom(A), first-unfrozen-geler(L, B).
I-choice(A+L,  A)       ⇐ not is-geler-atom(A), all-geler-frozen(L)

I-choice(geler(X,B)+L,  A)        ⇐ L-var(X) ,   I-choice(L,  A)

I-choice(geler(X, A)+L,  A)       ⇐ not L-var(X)

first-unfrozen-geler(L,A)         ⇐ conc(L1,  geler(X,  A)+L2,  L),
                                    all-geler-frozen(L1) ,
                                    not L-var(X)
```

```
all-geler-frozen(true)           ⇐ .

all-geler-frozen(A+L)            ⇐ not is-geler-atom(A),  all-geler-frozen(L)

all-geler-frozen(geler(X, A)+L)⇐ L-var(X),  all-geler-frozen(L)

is-geler-atom(A)                 ⇐ A= geler(_,_)
```

All frozen goals are unfrozen following the order of freezing. **conc** is the list concatenation on goal lists.

expand(P,T_1-N_1,A,T_2-N_2) then if P is a program, N_1 the current node of T_1, A the choosen literal in the goal of N_1 then T_2 is T_1 where N_1 has as many sons as there are unifiable heads with A in P (in all cases $N_2 = N_1$).

```
expand(P,  N-N,  A,  N-N)              ⇐ N = node(I, _),
                                          packet(P, I, A, nil)

expand(P, N-N,  A, mk-tree(N, F)-N)    ⇐ N = node(I, _),
                                          packet(P, I, A, Q), not Q = nil ,
                                          buildlist(Q, N, A, F,o)

expand(P, mk-tree(N, F1)-N1, A, mk-tree(N, F2)-N2)⇐ expand(P,  F1-N1,  A, F2-N2)

expand (P,  A1.L-N1,  A,  A2 .L-N2)    ⇐ expand(P,  A1-N1,  A,  A2-N2)

expand(P,  A.L1-N1,  B,  A.L2-N2)      ⇐ expand(P,  L1-N1,  B,  L2-N2)
```

buildlist(Q,N,A,F,I) then if Q is a packet, N a node, A a chosen literal in the goal of N, I a list of integers then F is the list of new sons of N.

```
buildlist(nil,  N,  A,  nil,  I)         ⇐

buildlist(clause(H, C).Q,  N, A, F, I)   ⇐ not L-unifiable(H, A) ,
                                            buildlist(Q,  N,  A,  F,  I)

buildlist(clause(H1, C1). Q,   N,   A, node(I.J, G2).F, I)
                                     ⇐ N = node(J, G),
                                        L-rename(I.J,clause(H1,C1), clause(H2;C2)),
                                        L-unify(H2,  A,  S),
                                        flagcut(C2,  I.J,  C3),
                                        replace(G,  A,  C3,  G1)
                                        L-instance(G1,  S,  G2),
                                        buildlist(Q, N, A, F,  succ (I))
```

replace(G,A,C,G_1) then if G is a goal, A the chosen literal in G, C a goal then G_1 is G in which A has been replaced by C.
(Note that replace depends on the I-strategy).

```
replace(A.L1, A, L2, L3)      ⇐ conc(L2, L1, L3)
```

flagcut(G_1,I,G_2) then if G_1 is a goal and I a list of integers then G_2 is G_1 in which all predicates cut have been flagged by I.

```
flagcut(true, I, true)        ⇐
flagcut(! + L1, I, !(I)+L2)   ⇐ flagcut(L1, I, L2)
flagcut(A+L1, I, A+L2)        ⇐ not A = !, flagcut(L1, I, L2)
```

treat-cut(T_1-N_1,T_2-N_2,J) then if N_1 is the current node of T_1, J a list of integers and the chosen literal in the goal of N_1 is a cut then T_2 is T_1 in which all brothers of father node of N_1 whose goal contains the same cut have been suppressed and a new son added to N_1 (N_1=N_2).

(depends on the I-strategy : here it is the standard one)

```
treat-cut(N-N, mk-tree(N,node(o.I,G2).nil)-N, I)        ⇐ N = node(I, G1),
                                                          tail(G1, G2)

treat-cut(mk-tree(N,F1)-N1, mk-tree(N,F2)-N2,I) ⇐         treat-cut(F1-N1, F2-N2, I)

treat-cut(T1.L-N1,T2.nil-N2,I)          ⇐ N1 in T1, cut-inside(I, T1),
                                          treat-cut(T1-N1, T2-N2, I)

treat-cut(T1.L-N1, T2.L-N2, I)          ⇐ N1 in T1, not cut-inside(I, T1),
                                          treat-cut(T1-N1, T2-N2, I)

treat-cut(T.L1-N1, T.L2-N2, I)          ⇐ N1 in L1, treat-cut(L1-N1, L2-N2, I)
```

cut-inside(I,T) then if I is a list of integers and T a search tree then there exists in the goal of the root of T a cut flagged by I.

```
cut-inside(I, node(J, G))               ⇐ cut-member(I, G)

cut-inside(I, mk-tree(node(J, G), F)) ⇐ cut-member(I, G)

cut-member(I, !(I)+L)                   ⇐
cut-member(I, A+L)                      ⇐ cut-member(I, L)
```

7)Discussion and conclusion:

We have presented a formal semantics for PROLOG dialects which seems to satisfy the criteria of a good formal specification method as given in the introduction:

- It is of high level because it contains no reference to any abstract machine. Also it does not use any too low level programming language. The logical notions it uses are simple and known by most PROLOG programmers.

- It can be used to describe with the same formalism most of the existing PROLOG dialects. Without major difficulties it is possible to take into account the following features :

  . cut
  . geler(or freeze), wait (any non standard deterministic strategy)
  . assert, retract
  . constraints, dif, delayed negation
  . infinite terms : Rational infinite terms can be added to the language L whose treatment can be described in a logical specification using finite terms only. Thus even if the described dialect containing rational infinite terms does not have any logical semantics (in the sense of section 2), the specification still has one.
  . assignment, global variables
  . escape or block mechanisms
  . arithmetics and other built-in predicates. These are in fact not described by clauses in a program, however they belong to the described language L and we want to include their description in the same formalism. It is partially possible. As an example we give here a description of a reversible built-in predicate **plus** using a semantical predicate **L-eval** with two arguments : some goal (chosen atom) and the "computed" answer substitution S.

```
L-eval(plus(X, Y, Z), S)    ⇐ ℕ-number(X),
                               ℕ-number(Y),
                               L-var(Z),
                               ℕ-plus(X,Y,Z1),
                               L-unify(Z, Z1, S)

L-eval(plus(X, Y, Z), S )   ⇐ ℕ-number(X),
                               ℕ-number(Z),
                               L-var(Y),
                               ℕ-plus(X,Y1,Z),
                               L-unify(Y, Y1, S)

L-eval(plus(X, Y, Z), S)    ⇐ ℕ-number(Y),
                               ℕ-number(Z),
                               L-var(X),
                               ℕ-plus(X1,Y,Z),
                               L-unify(X, X1, S).

L-eval(plus(X, Y, Z), Ø)    ⇐ ℕ-number(X),
                               ℕ-number(Y),
                               ℕ-number(Z),
                               ℕ-plus(X,Y,Z).
```

This description uses an undescribed predicate **ℕ-number**. Such predicate will be supposed to have a clear denotation which can be easily defined under the assumption that the domain ℕ of the natural integer is well known enough. Thus the denotation of **ℕ-number** contains all atoms **ℕ-number**(i) where i is an integer of ℕ, **ℕ-plus** is as in section 3.

- As the formal definition is a pure logic program the validation methods (correctness, completeness, ... [DF 86c]) can be applied to validate the formal definition itself.
- As the formal definition is a logic program included in every dialect, it should be rather simple to derive from the specification various validation tools like a runnable specification. It should be noted that a runnable specification should use built-in predicates with a standard l-strategy and implement the negation as failure. It is thus necessary to show specific properties preserving correctness, completeness and termination in some cases of the formal specification. However it will be possible to improve the specification by debugging also.

This kind of formal definition can be compared with other logical semantics of logic programming such as in [Mos 81] and [MR 86]. Moss's approach describes proof-trees construction in place of search-trees construction. It consists in giving a validating program (i.e. a PROLOG program simulating an interpreter) and thus it does not take into account for example the order in which the answers are computed. Martelli and Rossi approach can be considered as a runnable logic specification of a denotational one and it uses a kind of abstract machine. Considering other kind of semantics like the denotational semantics of [JM 84, Fra 85, Nor 86] , we could say that our semantics is less concise but easier to read, especially for people knowing some PROLOG dialects. Moreover it introduces the original notion of "comments" which are formalized as a part of the specification.

We have shown how a logic program of about sixty clauses could model the formal semantics of logic programs with cut and a non standard strategy. The size of a complete description of a dialect or a standard would increase in proportion with the number of considered primitives, but preserving the modularity and the clarity of the formal definition. It can be improved also by choosing simpler data structures as ramifications [Fer 85]. A draft proposal for PROLOG standard has been now presented with about 500 clauses describing 60 built-in predicates [DR 87]. Some questions, as input/output or error handling, have not been investigated here. Some adaptation of the presented semantic model would be necessary, but are still possible by using the same level of abstraction. This have been done in [DR 87].

ACKNOWLEDGEMENTS

We are indepted to Gilles Richard for comments on the earlier drafts which helped to improve this presentation, and to Chris Moss for helpful discussions.

BIBLIOGRAPHY

[AvE 82] K. R. Apt, M. H. Van Emden : Contributions to the theory of logic programming. JACM V29, N° 3, Jul. 1982 pp 841-862.

[ABW 86] K.R. Apt, H. Blair, A. Walker: Toward a Theory of Declarative Knowledge. LITP res. report 86-10, Fev 1986.

[Cla 79] K. L. Clark : Predicate Logic as a Computational Formalism. Res. Mon. 79/59 TOC. Imperial College. Dec. 79.

[Col 82] A. Colmerauer : Prolog II: Manuel de reference et modele theorique, GIA, Univ. of Marseille, 1982.

[Del 86] J.P. Delahaye : Sémantique logique et dénotationnelle des interpréteurs PROLOG. Note IT n° 84. University of Lille. 1986.

[DF 86a] P. Deransart, G. Ferrand: Initiation a PROLOG, Concepts de base. Pub. du Lab. d'Informatique, Univ. of Orleans, June 1986.

[DF 86b] P. Deransart, G. Ferrand: An Operational Formal Definition of PROLOG. A note for the AFNOR-BSI group on PROLOG normalization. 10/05/86, BSI PS/112.

[DF 86c] P. Deransart, G. Ferrand : Logic Programming, Methodology and Teaching. Actes du Seminaire 1986, CNET Trégastel, May 1986, pp75-90 (english version BSI, PS 127).

[DF 87a] P. Deransart, G. Ferrand : Programmation en Logique avec négation : présentation formelle. Publication du Laboratoire d'Informatique n° 87-3 (June 1987), University of Orléans.

[DF 87b] P. Deransart, G. Ferrand : An operational Formal Description of PROLOG. 4th Symposium on Logic Programming, August 31-September 4, 1987, San Francisco pp 162-172.

[DM 85] P. Deransart, J. Maluszynski : Relating logic Programs and Attribute Grammars. Journal of Logic Programming 1985, 2, pp 119-155.

[DR 87] P. Deransart, G. Richard : The formal specification of PROLOG standard-draft-BSI PS/198, AFNOR F41, March 1987, Draft 2, August 1987.

[Fer85] G. Ferrand : Error Diagnosis in Logic Programming, an Adaptation of E.Y. Shapiro's Method, RR375, INRIA Rocq. Mars 1985, and Journal of Logic Programming 1987, 49, pp 177-198.

[Fer86] G. Ferrand: A reconstruction of Logic Programming with Negation, Publication du Laboratoire d'Informatique n° 86-5 (dec. 1986), University of Orleans.

[Fra 85] G. Frandsen : A Denotational Semantics for Logic Programming. DAIMI PB 201, Aarhus University, Nov 1985.

[Gal 86] J.H. Gallier : Logic for computer Science, Harper & Row, 1986.

[JM 84] N. D. Jones, A. Mycroft : Stepwise Development of Operational and Denotational Semantics for Prolog. Proc. 1984 Int. Symp. on Logic Programming, Atlantic City, N.J., 1984.

[Kee 85] R.A. O'Keefe : A Formal Definition of Prolog. Univ. of Auckland, BSI PS/22.

[Llo 84] J. W Lloyd : Foundations of Logic Programming. Springer Verlag, Berlin, 1984.

[Mos 81] C.D.S. Moss : The Formal Description of Programming Languages using Predicate Logic. Imperial College. DCS-July 1981, (BSI PS/37).

[MR 86] A. Martelli, G. Rossi : On the Semantics of Logic Programming Languages. Third Int. Conf. on Logic Programming, London, Jul. 1986. LNCS 225, pp 327-334.

[Nor 86] N.D. North: PROLOG A denotational Definition. National Physical Laboratory, BSI-IST/5/15, PS141, Sept 1986.

[PWB 84] F. Pereira, D. Warren, D. Bowen, L. Byrd, L. Pereira : C-Prolog User's Manual. SRI International, Calif., Fev 1984.

[Rob 65] J. A. Robinson : A machine oriented logic based on the resolution principle. JACM 12, 1.

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