Introduction to prolog III

Article *in* Communications of the ACM · July 1990

DOI: 10.1145/79204.79210 · Source: DBLP

1 author:

Alain Colmerauer
Aix-Marseille Université

An Introduction to Prolog III

Alain Colmerauer
Professor at Aix-Marseille II University
Artificial Intelligence Group (GIA)
CNRS Laboratory 816
Faculté des Sciences de Luminy, Case 901
70 route Léon Lachamp,
13288 Marseille Cedex 9

Abstract. The Prolog III programming language extends Prolog by redefining the fundamental process at its heart : unification. Into this mechanism, Prolog III integrates refined processing of trees and lists, number processing, and processing of two-valued Boolean algebra. We present the specification of this new language and illustrate its capabilities by means of varied examples. We also present the theoretical foundations of Prolog III, which in fact apply to a whole family of programming languages. The central innovation is to replace the concept of unification by the concept of constraint solving.

Résumé. Le langage de programmation Prolog III est une extension de Prolog au niveau de ce qu'il a de plus fondamental, le mécanisme d'unification. Il intègre dans ce mécanisme un traitement plus complet des arbres et des listes, un traitement numérique et un traitement de l'algèbre de Boole à deux valeurs. Nous présentons ici les spécifications de ce nouveau langage et illustrons ses possibilités au moyen d'exemples variés. Nous présentons aussi le modèle théorique de Prolog III qui, en fait, s'applique à toute une famille de langages de programmation. L'idée essentielle est de remplacer la notion d'unification par celle de résolution de contraintes.

24/03/00

Contents

INTRODUCTION

Prolog was initially designed to process natural languages. Its application in various problem solving areas has brought out its qualities, but has also made clear its limits. Some of these limitations have been overcome as a result of more and more efficient implementations and ever richer environments. The fact remains, however, that the core of Prolog, namely, Alan Robinson's unification algorithm [22], has not fundamentally changed since the time of the first Prolog implementations, and is becoming less and less significant compared to the ever-increasing number of external procedures as, for example, the procedures used for numerical processing. These external procedures are not easy to use. Their invocation requires that certain parameters are completely known, and this is not in line with the general Prolog philosophy that it should be possible anywhere and at any time to talk about an unknown object x.

In order to improve this state of affairs, we have fundamentally reshaped Prolog by integrating at the unification level : (1) a refined manipulation of trees, including infinite trees, together with a specific treatment of lists, (2) a complete treatment of two-valued Boolean algebra, (3) a treatment of the operations of addition, subtraction, multiplication by a constant and of the relations <, =, >, =, (4) the general processing of the relation ?. By doing so we replace the very concept of unification by the concept of constraint solving in a chosen mathematical structure. By mathematical structure we mean here a domain equipped with operations and relations, the operations being not necessarily defined everywhere.

The result of incorporating the above features into Prolog is the new programming language Prolog III. In this paper[1] we establish its foundations and illustrate its capabilities using representative examples. These foundations, which in fact apply to a whole family of "Prolog III like" programming languages, will be presented by means of simple mathematical concepts without explicit recourse to first-order logic.

The research work on Prolog III is not an isolated effort; other research has resulted in languages whose design shares features with Prolog III. The CLP(R) language developed by J. Jaffar and S. Michaylov [19] emphasizes real number processing, whereas the CHIP language developed by the team led by M. Dincbas [13] emphasizes processing of Boolean algebra and pragmatic processing of integers and elements of finite sets. Let us also mention the work by J. Jaffar et J-L. Lassez [18] on a general theory of "Constraint Logic Programming". Finally, we should mention Prolog II, the by now well-established language which integrates infinite trees and the ? relation, and

---

[1] A very preliminary version of this paper has appeared in the *Proceedings of the 4th Annual ESPRIT Conference*, Brussels, North Holland, pp. 611-629, 1987.

whose foundations [8,9] were already presented in terms of constraint solving. From a historical point of view, Prolog II can thus be regarded as the first step towards the development of the type of languages discussed here.

THE STRUCTURE UNDERLYING PROLOG III

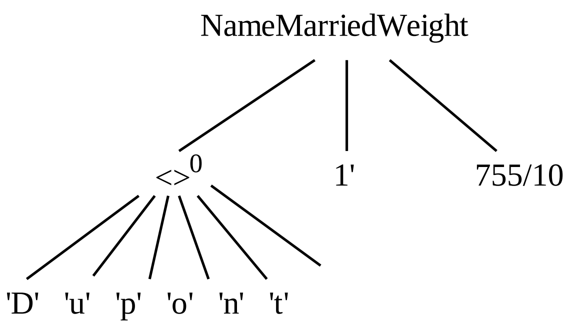
We now present the particular structure which is the basis of Prolog III and specify the general concept of a structure at the same time. By *structure* we mean a triple (D, F, R) consisting of a *domain* D, a set F of *operations* and a set of relations on D.

Domain

The domain D of a structure is any set. The domain of the structure chosen for Prolog III is the set of *trees* whose nodes are labeled by one of the following :

(1) identifiers,
(2) characters,
(3) Boolean values, 0' and 1',
(4) real numbers,
(5) special signs $<>^{\alpha}$, where α is either zero or a positive irrational number.

Here is such a tree :



The branches emanating from each node are ordered from left to right; their number is finite and independent of the label attached to the node. The set of nodes of the tree can be infinite. We do not differentiate between a tree having only one node and its label. Identifiers, characters, Boolean values, real numbers and special signs $<>^{\alpha}$ will therefore be considered to be particular cases of trees.

By *real numbers* we mean perfect real numbers and not floating point numbers. We make use of the partition of the reals into two large categories, the rational numbers, which can be represented by fractions (and of which the integers are a special case) and the irrational numbers (as for example π and $\sqrt{2}$) which no fraction can represent. In fact, the machine will compute with rational numbers only and this is related to an essential property of the constraints that can be employed in Prolog III : if a variable is sufficiently constrained to represent a unique real number then this number is necessarily a rational number.

A tree a whose initial node is labeled by $<>^{\alpha}$ is called a *list a*nd is written

$$<a_1,...,a_n>^{\alpha},$$

where $a_1...a_n$ is the (possibly empty) sequence of trees constituting the immediate daughters of a. We may omit α whenever α is zero. The *true* lists are those for which α is zero : they are used to represent sequences of trees (the sequence of their immediate daughters). Lists in which α is not zero are *improper* lists that we have not been able to exclude : they represent sequences of trees (the sequence of their immediate daughters) completed at their right by something unknown of length α. The *length* $|a|$ of the list a is thus the real $n+\alpha$. True lists have as their length a non-negative integer and improper lists have as their length a positive irrational number. The list <> is the only list with length zero, it is called the *empty list*. We define the operation of *concatenation* on a true list and an arbitrary list by the following equality :

$$<a_1,...,a_m>^0 \bullet <b_1,...,b_n>^{\alpha} = <a_1,...,a_m,b_1,...,b_n>^{\alpha}.$$

This operation is associative, $(a \bullet a') \bullet b = a \bullet (a' \bullet b)$, and the empty list play the role of the neutral element, $a \bullet <> = a$ et $<> \bullet b = b$. We observe that for any list b there exists one and only one truer list a and one and only one real α such that

$$b = a \bullet <>^{\alpha}.$$

This list a is called the *prefix* of b and is written $\lfloor b \rfloor$.

Operations

Let D^n denote the set of tuples $a_1...a_n$ constructed on the domain D of a structure. An *n-place operation f* is a mapping from a subset E of D^n to D,

$$f : a_1...a_n \mapsto f\, a_1...a_n .$$

Note that if E is strictly included in D^n, the operation f is partial; it is not defined for all tuples of size *n*. Note also that in order to be systematic the result of the operation is written in prefix notation. The 0-place operations are simply mappings of the form

$$f: \Lambda \mapsto f,$$

where Λ is the empty tuple; they are also called *constants* since they can be identified with elements of the domain.

As far as the chosen structure is concerned here is first of all a table listing the operations which belong to F. In this table we introduce a more graceful notation than the prefix notation.

Constants

id	:	$\Lambda \mapsto$ id,
'c'	:	$\Lambda \mapsto$ 'c',
0'	:	$\Lambda \mapsto$ 0',
1'	:	$\Lambda \mapsto$ 1',
q	:	$\Lambda \mapsto$ q,
$<>^0$	:	$\Lambda \mapsto$ <>,
$c_1 ... c_m$	:	$\Lambda \mapsto$ "$c_1 ... c_m$".

Boolean operations

$\neg$	:	$b_1 \mapsto \neg b_1$,
$\Rightarrow$	:	$b_1 b_2 \mapsto b_1 \Rightarrow b_2$,
$\vee$	:	$b_1 b_2 \mapsto b_1 \vee b_2$,
$\supset$	:	$b_1 b_2 \mapsto b_1 \supset b_2$,
$+$	:	$b_1 b_2 \mapsto b_1 + b_2$.

Numerical operations

$+^1$	:	$r_1 \mapsto +r_1$,
$-^1$	:	$r_1 \mapsto -r_1$,
$+^2$	:	$r_1 r_2 \mapsto r_1 + r_2$,
$-^2$	:	$r_1 r_2 \mapsto r_1 - r_2$,
q$\times$	:	$r_1 \mapsto$ q$\times r_1$,
/q'	:	$r_1 \mapsto r_1$/q'.

List operations

$\|\|$	:	$l_1 \mapsto \|l_1\|$,
$<,>^m$	:	$a_1 ... a_m \mapsto <a_1, ..., a_m>$,
$a_1 ... a_n \bullet$	:	$l_1 \mapsto <a_1, ..., a_n> \bullet l_1$.

General operations

$()^{n+2}$	:	$e_1 a_2 ... a_{n+2} \mapsto e_1(a_2, ..., a_{n+2})$,
[]	:	$e_1 l_2 \mapsto e_1[l_2]$.

Here *id* designates an identifier, c and c_i a character, q et q' rational numbers represented by fractions (or integers), with q' not zero, m a positive integer, n a non-negative integer and a_i an arbitrary tree. The result of the different operations is defined only if b_i is a Boolean value, r_i a real number, l_i a list and e_i a label not of the form $<>^\alpha$.

To each label corresponds a constant, with the exception of irrational numbers and labels of the form $<>^{\alpha}$, where α is not zero. The constant "$c_1...c_m$" designates the true list whose immediate daughters make up the sequence of characters 'c_1' ... 'c_m'. The operations $\neg$, $\Rightarrow$, $\vee$, $\supset$, $+$, correspond to the classical Boolean operations when they are defined. The operations $\pm^1$, $\pm^2$, $q\times$, when they are defined, are the 1-place $\pm$, the 2-place $\pm$, multiplication by the constant q (when this does not lead to confusion we may omit the sign $\times$) and division by the constant q'. By $|l_1|$ we designate the length of the list l_1. By $<a_1,...,a_m>$ we designate the true list whose immediate daughters make up the sequence $a_1,...,a_m$. The operation $a_1...a_n\bullet$ applied to a list l_1 consists in concatenating the true list $<a_1,...,a_n>$ to the left of l_1. By $e_1(a_2,...,a_{n+2})$ we designate the tree consisting of an initial node labeled e_1 and the sequence of immediate daughters $a_2,...,a_{n+2}$. By $e_1[l_2]$ we designate the tree consisting of an initial node labeled e_1 and of the sequence of immediate daughters of the list l_2.

We note the following equalities (provided the different operations used are indeed defined) :

$$"c_1...c_m" \;=\; <'c_1', \;...\; ,'c_m'>$$

$$a_0(a_1,...,a_m) \;=\; a_0[<a_1,...,a_m>].$$

Using the constants and the operations we have introduced, we can represent our previous example of a tree by

NameMarriedWeight("Dupont", 1', 755/10)

or by

NameMarriedWeight[<<'D','u'>•"pont", 0'∨1', 75+1/2>].

Relations

Let D^n again denote the set of tuples $a_1...a_n$ constructed on the domain D of a structure. An *n-place relation r* is a subset E of D^n to D. To express that the tuple $a_1...a_n$ is in the relation r we write

$$r\ a_1...a_n\ .$$

With respect to the structure chosen for Prolog III, here are the relations contained in F.

We also introduce a more graceful notation than the prefix notation.

One-place relations

id : a_1 : id,
char : a_1 : char,
bool : a_1 : bool,
num : a_1 : num,
irint : a_1 : irint,
list : a_1 : list,
leaf : a_1 : leaf.

Identity relations

$=$: $a_1 = a_2$,
? : a_1 ? a_2.

Boolean relations

$\Rightarrow$: $a_1 \Rightarrow a_2$.

Numerical relations

$<$: $a_1 < a_2$,
$>$: $a_1 > a_2$,
$=$: $a_1 = a_2$,
$=$: $a_1 = a_2$,

Approximated operations

$/^3$: $a_3 \doteq a_1 \dot{/} a_2$,
$\times^{n+1}$: $a_{n+1} \doteq a_1 \times \ldots \times a_n$,
$\bullet^{n+1}$: $a_{n+1} \doteq a_1 \dot{\bullet} \ldots \dot{\bullet} a_n$.

Here n designates an integer greater than 1 and a_i an arbitrary tree. The relations id, char, bool, num, irint, list and leaf are used to specify that the tree a_1 is an identifier, a character, a Boolean value, a real number, an integer or irrational number, a list, a label not of the form $<>^{\alpha}$. The relations = et ? correspond of course to the equality and inequality of trees. The pair of trees $a_1 a_2$ is in the relation $\Rightarrow$ only if a_1 et a_2 are Boolean values and if a_1 = 1' entails that a_2 = 1'. The pair of trees $a_1 a_2$ is in relation <, >, =, = only if it is a pair of reals in the corresponding classical relation.

We use the relation $/^3$ to approximate division and write

$$a_3 \doteq a_1 \,\dot{/}\, a_2$$

to express, on the one hand, that a_1, a_2 and a_3 are real numbers, with a_2 not equal to zero and, on the other hand, that if at least one of the reals a_2 et a_3 is rational, it is true that

$$a_3 = a_1/a_2.$$

We use the relations ∞^{n+1}, with $n = 2$, to approximate a series of multiplications and write

$$a_{n+1} \doteq a_1 \times \ldots \times a_n$$

to express, on the one hand, that the a_i's are real numbers and, on the other hand, that if the sequence $a_1 \ldots a_n$ contains n or $n-1$ rationals numbers, it is true that

$$a_{n+1} = a_1 \times \ldots \times a_n.$$

We use the relations $\bullet^{n+1}$, with $n = 2$, to approximate a series of concatenations and write

$$a_{n+1} \doteq a_1 \,\dot{\bullet}\, \ldots \,\dot{\bullet}\, a_n$$

to express that in all cases the a_i's are lists such that

$$|a_{n+1}| = |a_1| + \ldots + |a_n|$$

and that, according as the element $a_1 \bullet \ldots \bullet a_n$ is, or is not defined,

$$a_{n+1} = a_1 \bullet \ldots \bullet a_n$$

or

$$a_{n+1} \text{ is of the form } \lfloor a_1 \bullet \ldots \bullet a_k \rfloor \bullet b,$$

where b is an arbitrary list and k is the largest integer such that the element $a_1 \bullet \ldots \bullet a_k$ is defined.

We recall that $a_1 \bullet \ldots \bullet a_k$ is defined only if the lists a_1, ..., a_{k-1} are all true lists. We also recall also that $\lfloor a \rfloor$ designates the prefix of a, that is to say, the true list obtained by replacing the initial label $<>^{\alpha}$ of a with the label $<>^{0}$.

Terms and constraints

Let us suppose that we are working in a structure (D, F, R) and let V be a *universal* set of variables, given once and for all, used to refer to the elements of its domain D . We will assume that V is infinite and countable. We can now construct syntactic objects of two kinds, terms and constraints. *Terms* are sequences of juxtaposed elements from V≈F of one of the two forms,

$$x \quad \text{or} \quad f\, t_1 \ldots t_n,$$

where x is a variable, f an n-place operation and where the t_i's are less complex terms. *Constraints* are sequences of juxtaposed elements from V≈F≈R of the form

$$r\, t_1 \ldots t_n,$$

where r is an n-place relation and the t_i's are terms. We observe that in the definition of terms we have not imposed any restriction on the semantic compatibility between f and the t_i 's. These restrictions, as we will see, are part of the mechanism which takes a term to its "value".

We introduce first the notion of an *assignment* σ to a subset W of variables : such an assignment is simply a mapping from W into the domain D of the structure. This mapping σ extends naturally to a mapping σ^* from a set T_σ of terms into D specified by

$$\sigma^*(x) \;=\; \sigma(x),$$
$$\sigma^*(f\, t_1 \ldots t_n) \;=\; f\, \sigma^*(t_1) \ldots \sigma^*(t_n).$$

The terms that are not members of T_σ are those containing variables not in W, and those containing partial operations not defined for the arguments $\sigma^*(t_i)$. Depending on whether a term t belongs or does not belong to T_σ the *value* of t under the assignment σ is defined and equal to $\sigma^*(t)$ or is not defined. Intuitively, the value of a term under an assignment is obtained by replacing the variables by their values and by evaluating the term. If this evaluation cannot be carried out, the value of the term is not defined for this particular assignment.

We say that the assignment σ to a set of variables *satisfies* the constraint $r\, t_1 \ldots t_n$ if the value $\sigma^*(t_i)$ of each term t_i is defined and if the tuple $\sigma^*(t_1) \ldots \sigma^*(t_n)$ is in the relation r, that is to say if

$$r \ \sigma^*(t_1)\ldots\sigma^*(t_n).$$

Here are some examples of terms associated with the structure chosen for Prolog III. Instead of using the prefix notation, we adopt the notations used when the different operations where introduced.

$$<x>\bullet y,$$
$$x[y],$$
$$<x>\bullet 10,$$
$$\mathrm{duo}(+x,\ x \vee y).$$

The first term represents a list consisting of an element x followed by the list y. The second term represents a tree, which is not a list, whose top node is labeled by x and whose list of immediate daughters is y. The value of the third term is never defined, since the concatenation of numbers is not possible. The value of the last term is not defined under any assignment, since x cannot be a number and a Boolean value at the same time.

Here are now some examples of constraints. Again we adopt the notations introduced together with the different Prolog III relations.

$$z = y - x,$$
$$x \wedge \neg y \Rightarrow x \vee z,$$
$$i+j+k = 10,$$
$$\neg x \ ? \ y+z,$$
$$\neg x \ ? \ y+x.$$

We observe that there exist assignments to $\{x, y, z\}$ which satisfy the next to the last constraint (for example $\sigma(x) = 0'$, $\sigma(y) = 2$, $\sigma(z) = 2$), but that there is no assignment which satisfies the last constraint (the variable x cannot be a number and a Boolean value at the same time).

Systems of constraints

Any finite set S of constraints is called a *system of constraints*. An assignment σ to the universal set V of variables which satisfies every constraint of S is a *solution* of S. If σ is a solution of S and W a subset of V, then the assignment σ' to W which is such that for every variable x in W we have $\sigma'(x) = \sigma(x)$ is called a *solution* of S *on* W. Two systems of constraints are said to be *equivalent* if they have the same set of solutions and are said to be *equivalent on* W if they have the same set of solutions on W.

We illustrate these definitions with some examples from our structure.

- The assignment σ to V where $\sigma(x)$ = 1' for every variable x is a solution of the system of constraints $\{x = y, y$? $0\}$, but it is not a solution of the system $\{x = y, +y$? $0\}$.
- The assignment σ to $\{y\}$ defined by $\sigma(y) = 4$ is a solution on $\{y\}$ of the system $\{x = y, y$? $0\}$.
- The systems $\{x = y, +y$? $0\}$ and $\{-x = -y, y$? $0\}$ are equivalent. Similarly, the system $\{1 = 1, x = x\}$ is equivalent to the empty constraint system.
- The systems $\{x = y, y = z, x$? $z\}$ and $\{x < z\}$ are not equivalent, but they are equivalent on the subset of variables $\{x, z\}$.

It should be noted that all solvable systems of constraints are equivalent on the empty set of variables, and that all the non-solvable systems are equivalent. By *solvable* system, we of course mean a system that has at least one solution.

The first thing Prolog III provides is a way to determine whether a system of constraints is solvable and if so, to solve the system. For example, to determine the number x of pigeons and the number y of rabbits such that together there is a total of 12 heads and 34 legs, the following query

$$\{x+y = 12,\ 2x+4y = 34\}\ ?$$

gives rise to the answer

$$\{x = 7,\ y = 5\}.$$

To compute the sequence z of 10 elements which results in the same sequence no matter whether 1,2,3 is concatenated to its left or 2,3,1 is concatenated to its right, it will suffice to pose the query

$$\{|z| = 10,\ \langle 1,2,3\rangle\bullet z \doteq z\bullet\langle 2,3,1\rangle\}\ ?$$

The unique answer is

$$\{z = \langle 1,2,3,1,2,3,1,2,3,1\rangle\}.$$

If in the query the list <2,3,1> is replaced by the list <2,1,3> there is no answer, which means that the system

$$\{|z| = 10,\ \ \langle 1,2,3\rangle \bullet z \doteq z \dot{\bullet} \langle 2,1,3\rangle\}$$

is not solvable. In these examples the lists are all of integer length and are thus true lists. As a result, the approximated concatenations behave like true concatenations.

In this connection, the reader should verify that the system

$$\{\langle 1\rangle \bullet z \doteq z \dot{\bullet} \langle 2\rangle\}$$

is solvable (it suffices to assign to z any improper list having no immediate daughters), whereas the system

$$\{|z| = 10,\ \ \langle 1\rangle \bullet z \doteq z \dot{\bullet} \langle 2\rangle\},$$

which constraints z to be a true list, is not solvable. The same holds for approximated multiplication and division. Whereas the system

$$\{z \doteq x \dot{\infty} y, x = 1, y = 1, z < 0\}$$

is solvable (because the approximated product of two irrational numbers is any number), the system

$$\{z \doteq x \dot{\infty} y, x = 1, y = 1, z < 0, y = 1\},$$

which constrains y to be a rational number, is not solvable.

Another example of the solving of systems is the beginning of a proof that God exists, as formalized by George Boole [4]. The aim is to show that "something has always existed" using the following 5 premises :
(1) Something is.
(2) If something is, either something always was, or the things that now are have risen out of nothing.
(3) If something is, either it exists in the necessity of its own nature, or it exists by the will of another Being.
(4) If it exists by the will of its own nature, something always was.

(5) If it exists by the the will of another being, then the hypothesis, that the things which now are have risen out of nothing, is false.

We introduce 5 Boolean variables with the following meaning :
a = 1' for " Something is",
b = 1' for " Something always was",
c = 1' for " The things which now are have risen from nothing",
d = 1' for " Something exists in the necessity of its own nature ",
e = 1' for "Something exists by the will of another Being".
The 5 premises are easily translated into the system

$$\{a = 1' \ \ a \Rightarrow b \vee c,\ \ a \Rightarrow d \vee e,\ \ d \Rightarrow b,\ \ e \Rightarrow \neg c\}$$

which when executed as a query produces the answer

$$\{a = 1',\ \ b = 1',\ \ d \vee e = 1',\ \ e \vee c = 1'\}.$$

One observes that the value b is indeed constrained to 1'.

After these examples, it is time to specify what we mean by "solving" a system S of constraints involving a set W of variables. Intuitively, this means that we have to find all the solutions of S on W. Because there may be an infinite set of such solutions, it is not possible to enumerate them all. What is however possible is to compute a system in *solved* form equivalent to S and whose "most interesting" solutions are explicitly presented. More precisely by system in *solved* form, we understand a solvable system such that, for every variable x, the solution of S on $\{x\}$ is explicitly given, whenever this solution is unique. One can verify that in the preceding examples the systems given as answers were all in solved form.

Before we end this section let us mention a useful property for solving systems of constraints in the chosen structure.

PROPERTY. If S a system of Prolog III constraints and W a set of variables, then the two following propositions are equivalent :
(1) for every x in W, there are several numerical solutions of S on $\{x\}$;
(2) there exists a numerical irrational solution S on W.

By numerical solution, or irrational numerical solution, on a set of variables, we understand a solution in which all the variables in this set have real numbers as values, or irrational numbers as values.

SEMANTICS OF PROLOG III LIKE LANGUAGES

On the basis of the structure we have chosen, we can now define the programming language Prolog III. As the method employed is independent of the chosen structure, we define in fact the notion of a "Prolog III like" language associated with a given structure. The only assumption that we will make is that the equality relation is included in the set of relations of the structure in question.

Meaning of a program.

In a Prolog III type language, a program is a definition of a subset of the domain of the chosen structure (the set of trees in the case of Prolog III). Members of this subset are called *admissible* elements. The set of admissible elements is in general infinite and constitutes - in a manner of speaking - an enormous hidden database. The execution of a program aims at uncovering a certain part of this database.

Strictly speaking, a program is a set of *rules*: Each rule has the form

$$t_0 \rightarrow t_1 ... t_n \text{ , S}$$

where n can be zero, where the t_i's are terms and where S is a possibly empty system of constraints (in which case it is simply absent). The meaning of such a rule is roughly as follows: "provided the constraints in S are satisfied, t_0 is an admissible element if t_1 and ... and t_n are admissible elements (or if $n = 0$) ". Here is such a set of rules; this is our first example of a Prolog III program. It is an improvement on a program which is perhaps too well-known, but which remains a useful pedagogical tool : the calculation of well-balanced meals [9].

```
LightMeal(h, m, d) →
    HorsDœuvre(h, i)
    MainCourse(m, j)
    Dessert(d, k),
    {i = 0, j = 0, k = 0, i+j+k = 10};

MainCourse(m, i) → Meat(m, i);
MainCourse(m, i) → Fish(m, i);

HorsDœuvre(radishes, 1) →;
HorsDœuvre(pâté, 6) →;

Meat(beef, 5) →;
Meat(pork, 7) →;

Fish(sole, 2) →;
Fish(tuna, 4) →;

Dessert(fruit, 2) →;
Dessert(icecream, 6) →.
```

The meaning of the first rule is: "provided the four conditions $i = 0$, $j = 0$, $k = 0$, $i+j+k = 10$ are satisfied, the triple h,m,d constitutes a light meal, if h is an hors-d'œuvre with calorific value i, if m is a main course with caloric value j and if d is a dessert with calorific value k ". The meaning of the last rule is: " Ice-cream is a dessert with calorific value 6 ".

Let us now give a precise definition of the set of admissible elements. The rules in the program are in fact rule schemas. Each rule (of the above form) stands for the set of *evaluated rules*

$$\sigma^*(t_0) \rightarrow \sigma^*(t_1)...\sigma^*(t_n)$$

obtained by considering all the solutions σ of S for which the values $\sigma^*(t_\mathrm{i})$ are defined. Each evaluated rule

$$a_0 \rightarrow a_1...a_n,$$

in which only elements a_i of the domain occur, can be interpreted in two ways:

(1) as a *closure property* of certain subsets E of the domain: if all of $a_1,...,a_n$ are members of the subset E , then a_0 is also is a member of E (when n = 0, this property states that a_0 is a member of E),
(2) as a *rewrite rule* which, given a sequence of elements of the domain beginning with a_0, sanctions the replacement of this first element a_0 by the sequence $a_1...a_n$ (when n = 0, this is the same as deleting the first element a_0).

According to which of the two above interpretations is being considered, we formulate one or the other of the following definitions:

DEFINITION 1. The set of *admissible* elements is the smallest subset of the domain (in the sense of inclusion) which satisfies all the closure properties stemming from the program.

DEFINITION 2. The *admissible* elements are the members of the domain which (considered as unary sequences) can be deleted by applying rewrite rules stemming from the program a finite number of times.

In [8,11] we show that the smallest subset in the first definition does indeed exist and that the two definitions are equivalent. Let us re-examine the previous program example. Here are some samples of evaluated rules:

```
......................
LightMeal(pâté,sole,fruit) →
        HorsDœuvre(pâté,6) MainCourse(sole,2) Dessert(fruit,2) ;
......................
......................
MainCourse(sole, 2) → Fish(sole,2) ;
......................
......................
HorsDœuvre(pâté,6) → ;
......................
......................
Fish(sole,2) → ;
......................
......................
Dessert(fruit,2) → ;
......................
```

If we consider these rules to be closure properties of a subset of trees, we can successively conclude that the following three subsets are sets of admissible elements,

{HorsDœuvre(pâté,6), Fish(sole,2), Dessert(fruit,2)},
{MainCourse(sole,2)},
{LightMeal(pâté,sole,fruit)}

and therefore that the tree

LightMeal(pâté,sole,fruit)

is an admissible element. If we take these evaluated rules to be rewrite rules, the sequence constituted solely by the last tree can be deleted in the following rewrite steps

LightMeal(pâté,sole,fruit) →
HorsDœuvre(pâté,6) MainCourse(sole,2) Dessert(fruit,2) →
MainCourse(sole,2) Dessert(fruit,2) →
Fish(sole,2) Dessert(fruit,2) →
Dessert(fruit,2) →,

which indeed confirms that the above is an admissible element.

Execution of a program

We have now described the information implicit in a Prolog III like program, but we have not yet explained how such a program is executed. The aim of the program's execution is to solve the following problem: given a sequence of terms $t_1 ... t_n$ and a system S of constraints, find the values of the variables which transform all the terms t_i into admissible elements, while satisfying all the constraints in S. This problem is submitted to the machine by writing the *query*

$$t_1 .... t_n \text{ , S ?}$$

Two cases are of particular interest. (1) If the sequence $t_1 ... t_n$ is empty, then the query simply asks wether the system S is solvable and if so to solve it. We have already seen examples of such queries. (2) If the system S is empty (or absent) and the sequence of terms is reduced to one term only, the request can be summarized as: "What are the values of the variables which transform this term into an admissible element?". Thus using the preceding program example, the query

LightMeal(*h*, *m*, *d*)?

will enable us to obtain all the triples of values for *h*, *m*, and *d* which constitute a light meal. In this case, the replies will be the following simplified systems :

{h = radishes, m = beef, d = fruit},
{h = radishes, m = pork, d = fruit},
{h = radishes, m = sole, d = fruit},
{h = radishes, m = sole, d = icecream},
{h = radishes, m = tuna, d = fruit},
{h = pâté, m = sole, d = fruit}.

The method of computing these answers is explained by introducing an abstract machine. This is a non-deterministic machine whose state transitions are described by these three formulas :

(1) $(\mathrm{W},\ t_0\, t_1 \ldots t_n\ ,\ \mathrm{S}),$

(2) $s_0 \to s_1 \ldots . s_m\ ,\ \mathrm{R}$

(3) $(\mathrm{W},\ s_1 \ldots s_m\, t_1 \ldots t_n\ ,\ \mathrm{S} \approx \mathrm{R} \approx \{t_0 = s_0\}).$

Formula (1) represents the state of the machine at a given moment. W is a set of variables whose values we want to determine, $t_0 t_1 \ldots t_n$ is a sequence of terms which we are trying to delete and S is a system of constraints which has to be satisfied. Formula (2) represents the rule in the program which is used to change the state. If necessary, the variables of (2) are renamed, so that none of them are shared with (1). Formula (3) is the new state of the machine after the application of rule (2). The transition to this new state is possible only if the system of constraints in (3) possesses at least one solution σ with respect to which all the values $\sigma^*(s_i)$ and $\sigma^*(t_i)$ are defined.

In order to provide an answer to the query given above, the machine starts from the initial state

$$(\mathrm{W},\ t_0 \ldots t_n,\ \mathrm{S}),$$

where W is the set of variables appearing in the query, and goes through all the states which can be reached by authorized transitions. Each time it arrives at a state containing the empty sequence of terms $\wedge$, it simplifies the system of constraints associated with it and presents it as an answer. This simplification can also be carried out on all the states it passes through.

Let us now reconsider our first program example, and apply this process to the query

$$\text{LightMeal}(h, m, d)?$$

The initial state of the machine is
$(\{h,m,d\},\ \text{LightMeal}(h,m,d),\ \{\})$.
By applying the rule
$\text{LightMeal}(h', m', d') \rightarrow$
$\text{HorsD\oe uvre}(h', i)\ \text{MainCourse}(m', j)\ \text{Dessert}(d', k),$
$\{i = 0, j = 0, k = 0, i+j+k = 10\}$
we proceed to the state
$(\{h,p,d\},\ \text{HorsD\oe uvre}(h',i)\ \text{MainCourse}(m',j)\ \text{Dessert}(d',k),$
$\{i=0, j=0, k=0, i+j+k=10, \text{LightMeal}(h,m,d)=\text{LightMeal}(h',m',d')\})$
which in turn simplifies to
$(\{h,p,d\},\ \text{HorsD\oe uvre}(h',i)\ \text{MainCourse}(m',j)\ \text{Dessert}(d',k),$
$\{i=0, j=0, k=0, i+j+k=10, h=h', p=p', d=d'\}),$
and to
$(\{h,p,d\},\ \text{HorsD\oe uvre}(h,i)\ \text{MainCourse}(m,j)\ \text{Dessert}(d,k),$
$\{i=0, j=0, k=0, i+j+k=10\})$.
By applying the rule
$\text{HorsD\oe uvre}(\text{pâté}, 6) \rightarrow$
and simplifying the result, we progress to the state
$(\{h,p,d\},\ \text{MainCourse}(p,j)\ \text{Dessert}(d,k),\ \{h=\text{pâté}, j=0, k=0, j+k=4\})$.
By applying the rule
$\text{MainCourse}(p', i) \rightarrow \text{Fish}(p', i)$
and simplifying the result, we proceed to the state
$(\{h,m,d\},\ \text{Fish}(m',i)\ \text{Dessert}(d,k),$
$\{h=\text{pâté}, j=0, k=0, j+k=4, m=m', j=i\})$.
which then simplifies to
$(\{h,m,d\},\ \text{Fish}(m,j)\ \text{Dessert}(d,k),\ \{h=\text{pâté}, j=0, k=0, j+k=4\})$.
By applying the rule
$\text{Fish}(\text{sole}, 2) \rightarrow$
we obtain
$(\{h,m,d\},\ \text{Dessert}(d,k),\ \{h=\text{pâté}, m=\text{sole}, k=0, k=2\})$.
Finally, by applying the rule
$\text{Dessert}(\text{fruit}, 2) \rightarrow$
We obtain
$(\{h,m,d\},\ \lambda,\ \{h=\text{pâté}, m=\text{sole}, d=\text{fruit}\})$.
We can conclude that the system

$$\{h = \text{pâté}, m = \text{sole}, d = \text{fruit}\}$$

constitutes one of the answers to the query.

To obtain the other answers, we proceed in the same way, but by using the other rules. In [11] we prove that this method is complete and correct. To be more exact, given the abstract machine M_P associated to a program P, we show that the following property holds.

PROPERTY. Let $\{t_1,...,t_n\}$ be a set of terms, S a system of constraints, and W the set of variables occurring in them. For any assignment σ to W, the following two propositions are equivalent :
(1) the assignment σ is a solution of S on W and each $\sigma^*(p_i)$ is an admissible element for P;
(2) starting from state (W, $\wedge$, S') the abstract machine M_P can reach a state of the form (W, $t_1...t_n$, S), where S' admits σ as solution on W.

It should be pointed out that there are a thousand ways of simplifying the states of the abstract machine and checking whether they contain solvable systems of constraints. So we should not always expect that the machine, which uses very general algorithms, arrives at the same simplifications as those that are shown above. In [11] we show that the only principle that simplifications must all conform to is that states of the abstract machine are transformed into equivalent states in the following meaning :

DEFINITION. Two states are *equivalent* if they have the form

$$(W, t_1...t_n , S) \text{ and } (W, t_1'...t_n', S'),$$

and if, by introducing n new variables $x_1,...,x_n$, the systems

$$S \approx \{x_1=t_1,...,x_n=t_n\} \text{ and } S' \approx \{x_1=t_1',...,x_n=t_n'\},$$

are equivalent on the subset of variables $W \approx \{x_1,...,x_n\}$.

TREATMENT OF NUMBERS

All that remains to be done is to illustrate the possibilities of Prolog III in connection with different program examples. We will consider one after the other the treatment of numbers, the treatment of Boolean values, the treatment of trees and lists and finally the treatment of integers.

Computing instalments

The first task is to calculate a series of instalments made to repay capital borrowed at a certain interest rate. We assume identical time periods between two instalments and an interest rate of 10% throughout. The admissible trees will be of the form :

InstalmentsCapital(x,c),

where x is the sequence of instalments necessary to repay the capital c with an interest rate of 10% between two instalments. The program itself is given by two rules :

```
InstalmentsCapital(<>, 0) →;
InstalmentsCapital(<i>•x, c) →
        InstalmentsCapital(x, (110/100)c–i);
```

The first rule expresses the fact that it is not necessary to pay instalments to repay zero capital. The second rule expresses the fact that the sequence of n+1 instalments to repay capital c consists of an instalment i and a sequence of n instalments to repay capital c increased by 10% interest, but the whole reduced by instalment i.

This program can be used in different ways. One of the most spectacular is to ask what value of i is required to repay $1000 given the sequence of instalments <i,2i, 3i> . All we need to do is to put the query

InstalmentsCapital(<i, 2i, 3i>, 1000) ?

to obtain the answer

{i = 207 + 413/641}.

Here is an abbreviated trace of how the computation proceeds. Starting from the initial state

({i}, InstalmentsCapital(<i,2i,3i>,1000), {}).

and applying the rule

InstalmentsCapital(<i'>•x,c) → InstalmentsCapital(x,(1+10/100)c–i')

we progress to the state

({i}, InstalmentsCapital(x,(1+10/100)c–i'),
{InstalmentsCapital(<i,2i,3i>,1000)=InstalmentsCapital (<i'>•x,c)}),

which simplifies to

({*i*}, InstalmentsCapital(*x*,(11/10)*c*-*i'*), {*i'*=*i*, *x*=<2*i*,3*i*>, *c*=1000}),

then to

({*i*}, InstalmentsCapital(<2*i*,3*i*>,1100–*i*), {}).

The reader can verify that when the same rule is applied two more times, we obtain, after simplification, the states

({*i*}, InstalmentsCapital(<3*i*>,1210–(31/10)*i*), {}),

({*i*}, InstalmentsCapital(<>,1331–(641/100)*i*), {}).

By applying the rule

InstalmentsCapital (<>,0) →

to the last state, we finally obtain

({*i*}, , {1331–(641/100)*i*=0}

which simplifies to

({*i*}, , {*i*=207+413/641 }).

Here again the reader should be aware that the simplifications presented above are not necessarily those the machine will perform.

Computing scalar products

As an example of approximated multiplication, here is a small program which computes the scalar product $x_1 \infty y_1 + \ldots + x_n \infty y_n$ of two vectors $<x_1,\ldots,x_n>$ and $<y_1,\ldots,y_n>$.

```
ScalarProduct(<>, <>, 0) →;
ScalarProduct(<x>•X, <y>•Y, u+z) →
        ScalarProduct(X, Y, z),
        {u ≐ x ´ y};
```

The query

ScalarProduct(<1,1>, *X*, 12) ScalarProduct(*X*, <2,4>, 34) ?

produces the answer

{*X* = <7,5>}.

Computing the periodicity of a sequence

This problem was proposed in [5]. We consider the infinite sequence of real numbers defined by

$$x_{i+2} = |x_{i+1}| - x_i$$

where x_1 and x_2 are arbitrary numbers. Our aim is to show that this sequence is always periodic and that the period is 9, in other words, that the sequences

$$x_1, x_2, x_3, \ldots \quad \text{and} \quad x_{1+9}, x_{2+9}, x_{3+9}, \ldots$$

are always identical.

Each of these two sequences is completely determined if its first two elements are known. To show that the sequences are equal, it is therefore sufficient to show that in any sequence of eleven elements

$$x_1, x_2, x_3, \ldots, x_{10}, x_{11}$$

we have

$$x_1 = x_{10} \quad \text{and} \quad x_2 = x_{11}.$$

To begin with, here is the program that enumerates all the finite sequences $x_1,x_2,\ldots,x_n$ which respect the rule given above :

```
Sequence(<+y, +x>) →;
Sequence(<z,y,x>•s) →
    Sequence(<y,x>•s)
    AbsoluteValue(y, y'), {z = y'–x};

AbsoluteValue(y, y) →, {y = 0};
AbsoluteValue(y, –y) →, {y < 0};
```

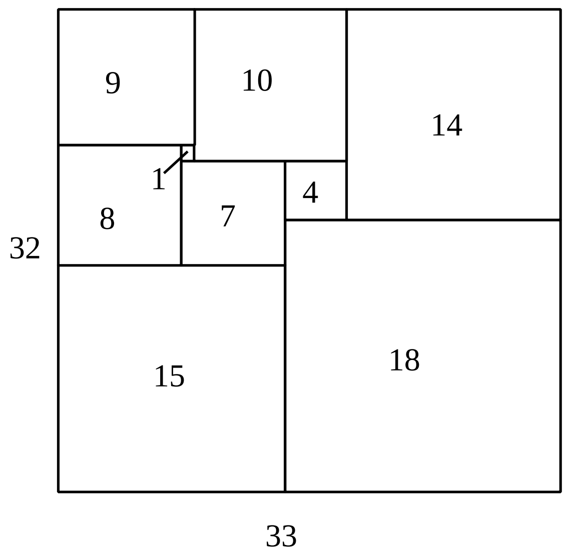
The + signs in the first rule constrain x and y to denote numbers. It will be observed that the sequences are enumerated from left to right, that is, trees of the form Sequence(s) are only admissible if s has the form $<x_n,\ldots,x_2,x_1>$. If we run this program by asking

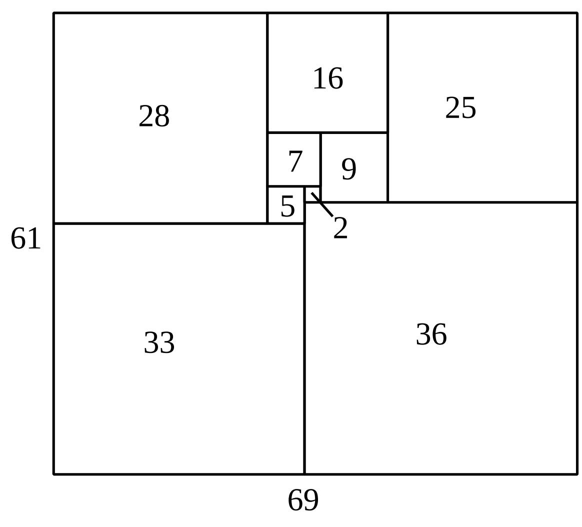
$$\text{Sequence}(s),\ \{|s| = 11,\ s \doteq w \bullet v \bullet u,\ |u| = 2,\ |w| = 2,\ u\ ?\ w\}\ ?$$

execution ends without providing an answer. From this we deduce that there is no sequence of the form $x_1,x_2,...,x_{10},x_{11}$ such that the subsequences x_1,x_2 and x_{10},x_{11} (denoted by u and v) are different, and therefore that in any sequence $x_1,x_2,...,x_{10},x_{11}$ we have indeed $x_1 = x_{10}$ and $x_2 = x_{11}$.

Computing a geometric covering

Here is a last example which highlights the numerical part of Prolog III. Given an integer n, we want to know whether it is possible to have n squares of different sizes which can be assembled to form a rectangle. If this is possible, we would in addition like to determine the sizes of these squares and of the rectangle thus formed. For example, here are two solutions to this problem, for n=9.





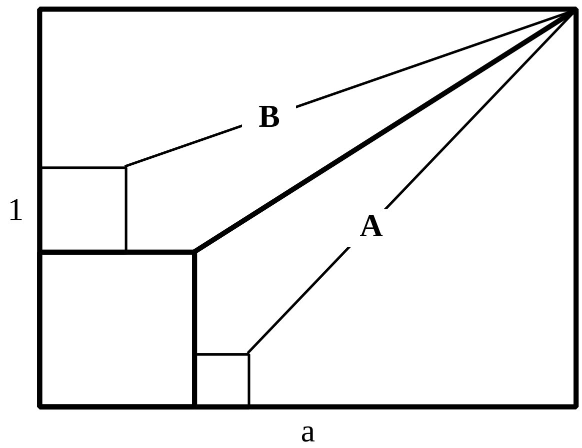
We will use a to denote the ratio between the length of the longest side of the constructed rectangle, and the length of its shortest side. Obviously, we can suppose that the length of the shortest side is 1, and therefore that the length of the longest side is *a*. Thus, we have to fill a rectangle having the size 1∞*a* with *n* squares, all of them different. With reference to the diagram below, the basis of the filling algorithm will consist of

(1) placing a square in the lower left-hand corner of the rectangle,

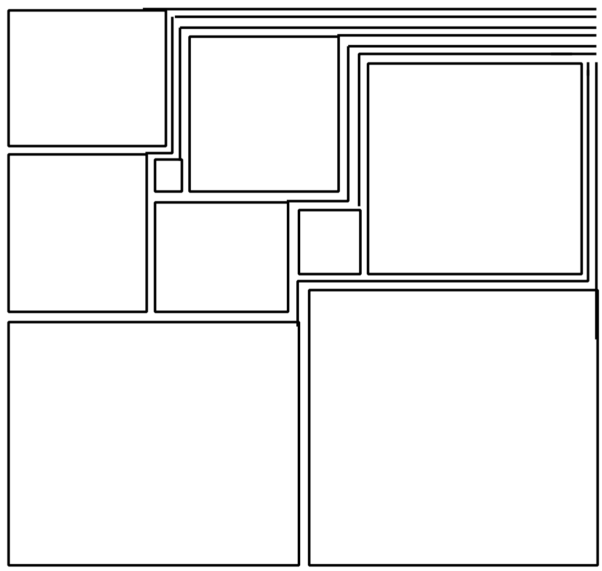
(2) filling zone A with squares,

(3) filling zone B with squares.

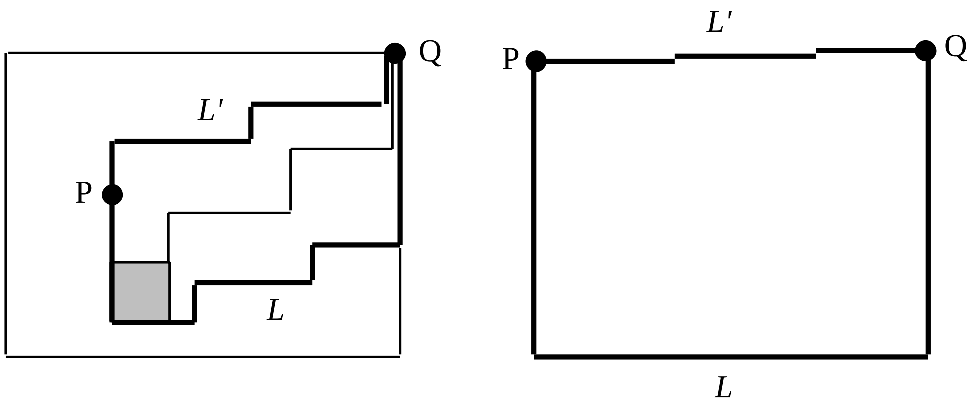
Provided zones A and B are not empty, they will be filled recursively in the same way: placing a square in the lower left-hand corner and filling two subzones.



The zones and subzones are separated by jagged lines in the shape of steps, joining the upper right corner of the squares and the upper right corner of the rectangle. These jagged lines never go downwards, and if several can be plotted to go from one point to another, the lowest one is the one which is considered. Here are for example all the separation lines corresponding to the first solution of the problem for n = 9 :



To be more precise, a zone or subzone has the form given in the left diagram below, whereas the entire rectangle is itself identified with the particular zone drawn on the right.



The zone is delimited below by a jagged line L joining a point P to a point Q, and above by a jagged line L' joining the same point P to the same point Q. Point P is placed anywhere in the rectangle to be filled, and Q denotes the upper right corner of the rectangle. These jagged lines are represented by alternating sequences of vertical and horizontal segments

$$v_0, h_1, v_1, \ldots, h_n, v_n,$$

where v_i denotes the length of a vertical segment, and h_i the length of a horizontal segment. The h_i's are always strictly positive. The v_i's are either zero, either positive to denote ascending segments, or negative to denote descending segments. The v_i's of the upper lines are never negative, and if a zone is not empty, only the first vertical segment v_0 in its lower line is negative.

If theses conventions are applied to the entire rectangle (right diagram above), the lower line L can be represented by the sequence -1,a,1 and the upper line L' by a sequence having the form $0,h_1,0,...,h_n,0$, where $h_1+...+h_n$ = a, and the h_i's are positive.

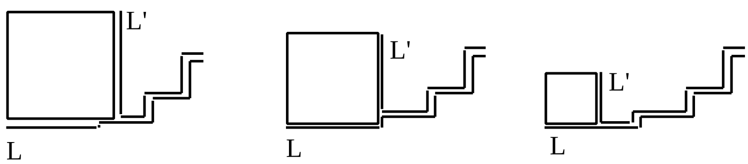
The heart of the program consists in admitting trees of the form

FilledZone(L, L', C, C')

only if the zone delimited below by L can be filled with squares and can be bounded above by L'. The squares are to be taken from the beginning of the list C, and C' has to be the list of squares which remain. We also need to introduce trees of the form

PlacedSquare(b, L, L')

which are admitted only if it is possible to place a square of size bxb at the very beginning of line L and if L' is the line making up the right vertical side of the square continued by the right part of L (see diagram below). In fact L denotes the lower line of a zone from which the first vertical segment has been removed. The diagram below shows the three cases that can occur and which will show up in three rules. Either the square overlaps the first step, which in fact was a pseudo step of height zero, or the square fits against the first step, or the square is not big enough to reach the first step.



The program itself is constituted by the following ten rules:

```
FilledRectangle(a, C) →
    DistinctSquares(C)
    FilledZone(<–1,a,1>, L, C, <>),
    {a = 1};

DistinctSquares(<>) →;
DistinctSquares(<b>•C) →
    DistinctSquares(C)
    OutOf(b, C),
    {b > 0};

OutOf(b, <>) →;
OutOf(b, <b'>•C) →
    OutOf(b, C),
    {b ? b'};

FilledZone(<v>•L, <v>•L, C, C) →,
    {v = 0};
FilledZone(<v>•L, L''', <b>•C, C'') →
    PlacedSquare(b, L, L')
    FilledZone(L', L'', C, C')
    FilledZone(<v+b,b>•L'', L''', C', C''),
    {v < 0};

PlacedSquare(b, <h,0,h'>•L, L') →
    PlacedSquare(b, <h+h'>•L, L'),
    {b > h};
PlacedSquare(b, <h,v>•L, <–b+v>•L) →,
    {b = h};
PlacedSquare(b, <h>•L, <–b,h–b>•L) →,
    {b < h};
```

The call to the program is made with the query

```
FilledRectangle(a, C), {|C| = n} ?
```

where *n*, the only known parameter, is the number of squares of different sizes that are to fill the rectangle. The program computes the possible size 1∞*a* of the rectangle (*a*=1) and the list C of the sizes of each of the *n* squares. The computation begins by executing the first rule, which at the same time constrains *a* to be greater than or equal to 1, creates *n* different squares (of unknown size) and starts filling the zone constituted by the entire rectangle. Even if the line L constituting the upper limit of this zone is unknown at the beginning, given that this line must join - without itself descending - two points at the same height, this line will necessarily be a horizontal line (represented by steps of height zero). If we ask the query

FilledRectangle(*a*, *C*), {|*C*| = *9*} ?

we obtain 8 answers. The first two

{*a* = 33/32, *C* = <15/32, 9/16, 1/4, 7/32, 1/8, 7/16, 1/32, 5/16, 9/32>},

{*a* = 69/61, *C* = <33/61, 36/61, 28/61, 5/61, 2/61, 9/61, 25/61, 7/61, 16/61>}.

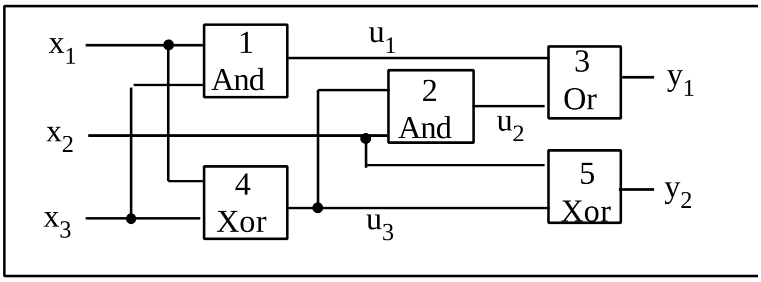
correspond to the two solutions we have drawn earlier. The other 6 answers describe solutions which are symmetrical to these two. In order to locate the positions of the various squares in the rectangle we can proceed as follows. One fills the rectangle using one after the other all the squares of the list C in their order of appearance. At each stage one considers all the free corners having the same orientation as the lower left corner of the rectangle and one chooses the rightmost one to place the square.

There is a vast literature concerning the problem that we have just dealt with. Let us mention to important results. It has been shown in [25] that for any rational number *a* = 1 there always exists an integer *n* such that the rectangle of size 1∞*a* can be filled with *n* distinct squares. For the case of *a* = 1, that is when the rectangle to be filled is a square, it has been shown in [14] that the smallest possible *n* is *n* = 21.

TREATMENT OF BOOLEANS VALUES

Computing faults

In this example we are interested in detecting the defective components in an adder which calculates the binary sum of three bits x_1, x_2, x_3 in the form of a binary number given in two bits y_1y_2. As we can see below, the circuit proposed in [16] is made up of 5 components numbered from 1 to 5: two *and* gates (marked And), one *or* gate (marked Or) and two *exclusive or* gates (marked Xor). We have also used three variables u_1, u_2, u_3 to represent the output from gates 1, 2 and 4.



We introduce 5 more Boolean variables d_i to express by d_i = 1' that "gate number i is defective". If we adopt the hypothesis that at most one of the five components has a defect, the program connecting the values x_i , y_i and d_i is :

```
Circuit(<x1,x2,x3>,  <y1,y2>,  <d1,d2,d3,d4,d5>) →
        AtMostOne(<d1,d2,d3,d4,d5>),
        {¬d1 => (u1 + x1∧x3),
        ¬d2  => (u2 + x2∧u3),
        ¬d3  => (y1 + u1∨u2),
        ¬d4  => (u3 + ¬(x1+x3)),
        ¬d5  => (y2 + ¬(x2+u3))};

AtMostOne(D) →
        OrInAtMostOne(D, d);

OrInAtMostOne(<>, 0') →;
OrInAtMostOne(<d>•D,  d∨e) →
        OrInAtMostOne(D, e),
        {d∧e = 0'};
```

In this program the admissible trees of the form

AtMostOne(D)

are those in which D is a list of Boolean elements containing at most one 1'. The admissible trees of the form

OrInAtMostOne(D, d)

are those in which D is a list of Boolean elements containing at most one 1' and where d is the disjunction of these elements.

If the state of the circuit leads us to write the query

Circuit(<1', 1', 0'>, <0', 1'>, <*d*1, *d*2, *d*3, *d*4, *d*5>)?

the diagnosis will be that component number 4 is defective :

$$\{ d1 = 0',\ d2 = 0',\ d3 = 0',\ d4 = 1',\ d5 = 0' \}.$$

If the state of the circuit leads us to write the query

Circuit(<1', 0', 1'>, <0', 0'>, <*d*1, *d*2, *d*3, *d*4, *d*5>) ?

the diagnosis will then be that either component number 1 or component number 3 is the defective one:

$$\{ d1 \vee d3 = 1',\ d1 \wedge d3 = 0',\ d2 = 0',\ d4 = 0',\ d5 = 0' \}.$$

Computing inferences

We now consider the 18 sentences of a puzzle due to Lewis Carroll [7], which we give below. Questions of the following type are to be answered : "what connection is there between being clear-headed, being popular and being fit to be a Member of Parliament?" or "what connection is there between being able to keep a secret, being fit to be a Member of Parliament and being worth one's weight in gold ?".

1. Any one, fit to be an M.P., who is not always speaking, is a public benefactor.

2. Clear-headed people, who express themselves well, have a good education.

3. A woman, who deserves praise, is one who can keep a secret.

4. People, who benefit the public, but do not use their influence for good purpose, are not fit to go into Parliament.

5. People, who are worth their weight in gold and who deserve praise, are always unassuming.

6. Public benefactors, who use their influence for good objects, deserve praise.

7. People, who are unpopular and not worth their weight in gold, never can keep a secret.

8. People, who can talk for ever and are fit to be Members of Parliament, deserve praise.

9. Any one, who can keep a secret and who is unassuming, is a never-to-be-forgotten public benefactor.

10. A woman, who benefits the public, is always popular.

11. People, who are worth their weight in gold, who never leave off talking, and whom it is impossible to forget, are just the people whose photographs are in all the shop-windows.

12. An ill-educated woman, who is not clear-headed, is not fit to go to Parliament.

13. Any, one, who can keep a secret and is not for ever talking, is sure to be unpopular.

14. A clear-headed person, who has influence and uses it for good objects, is a public benefactor.

15. A public benefactor, who is unassuming, is not the sort of person whose photograph is in every shop-window.

16. People, who can keep a secret and who who use their influence for good purposes, are worth their weight in gold.

17. A person, who has no power of expression and who cannot influence others, is certainly not a woman.

18. People, who are popular and worthy of praise, either are public benefactors ore else are unassuming.

Each of these 18 statements is formed from basic propositions and logical connectives. To each basic proposition corresponds a name, in the form of a character string, and a logical value represented by a Boolean variable. The information contained in the 18 statements can then be expressed in a single rule formed by a large head term, an empty body, and a sizeable constraint part :

```
PossibleCase(<
<a,"clear-headed">,
<b,"well-educated">,
<c,"constantly talking">,
<d,"using one's influence for good objects">,
<e,"exhibited in shop-windows">,
<f,"fit to be a Member of Parliament">,
<g,"public benefactors">,
<h,"deserving praise">,
<i,"popular">,
<j,"unassuming">,
<k,"women">,
<l,"never-to-be-forgotten">,
<m,"influential">,
<n,"able to keep a secret">,
<o,"expressing oneself well">,
<p,"worth one's weight in gold">>)      → ,
        {(f ∧ ¬c) ⇒ g,
        (a ∧ o) ⇒ b,
        (k ∧ h) ⇒ n,
        (g ∧ ¬d) ⇒ ¬f,
        (p ∧ h) ⇒ j,
        (g ∧ d) ⇒ h,
        (¬i ∧ ¬p) ⇒ ¬n,
        (c ∧ f) ⇒ h,
        (n ∧ j) ⇒ (g ∧ l),
        (k ∧ g) ⇒ i,
        (p ∧ c ∧ l) ⇒ e,
        (k ∧ ¬a ∧ ¬ b) ⇒ ¬f,
        (n ∧ ¬c) ⇒ ¬i,
        (a ∧ m ∧ d) ⇒ g,
        (g ∧ j) ⇒ ¬e,
        (n ∧ d) ⇒ p,
        (¬o ∧ ¬m) ⇒ ¬k,
        (i ∧ h) ⇒ (g ∨ j)};
```

To be able to deal with subcases, we introduce :

```
PossibleSubCase(x) →
        PossibleCase(y)
        SubSet(x, y);

SubSet(<>, y) →;
SubSet(<e>•x, y) →
        ElementOf(e, y)
        SubSet(x, y);

ElementOf(e, <e>•y) →;
ElementOf(e, <e'>•y) →
        ElementOf(e, y),  {e ? e' };
```

In order to compute the connection which exists between "clear-headed", "popular"and "fit to be a Member of Parliament" it suffices to write the query

```
PossibleSubCase(<
        <p,"clear-headed">,
        <q,"popular">,
        <r,"fit to be a Member of Parliament">>)?
```

The answer is the set of constraints

{*p* : bool, *q* : bool, *r* : bool},

which means that there is no connection between "clear-headed", "popular" and "fit to be a Member of Parliament".

To compute the connection which exists between "able to keep a secret", "fit to be a Member of Parliament" and "worth one's weight in gold" it suffices to write the query

```
PossibleSubCase(<
        <p,"able to keep a secret">,
        <q,"fit to be a Member of Parliament">,
        <r,"worth one's weight in gold">>) ?
```

The answer is

$\{p \wedge q \Rightarrow r\}$,

which means that persons who can keep a secret and are fit to be a Member of Parliament are worth their weight in gold.

In fact in these two examples of program execution we have assumed that Prolog III yields as answer very simplified solved systems, in particular, systems not containing superfluous Boolean variables. If this head not been the case, to show (as opposed to find) that persons who can keep a secret and are fit to be a Member of Parliament are worth their weight in gold, we would have had to pose the query

PossibleSubCase(<
<*p*,"able to keep a secret">,
<*q*,"fit to be a Member of Parliament">,
<*r*,"worth one's weight in gold">>),
$\{x = (p \wedge q \supset r)\}$?

and obtain a response of the form $\{x = 1', \ldots\}$ or obtain no answer to the query

PossibleSubCase(<
<*p*,"able to keep a secret">,
<*q*,"fit to be a Member of Parliament">,
<*r*,"worth one's weight in gold">>),
$\{(p \wedge q \supset r) = 0'\}$?

TREATMENT OF TREES AND LISTS

Computing the leaves of a tree

Here is first of all an example where we access labels and daughters of a tree by the operation []. We want to calculate the list of the leaves of a finite tree without taking into account the leaves labeled <>$^{\alpha}$. Here is the program :

```
Leaves(e[u], <e>) →,
        {u = <>};
Leaves(e[u], x) →
        Leaves(u, x),
        {u ? <>};
Leaves(<>, <>) →;
Leaves(<a>•u, z) →
        Leaves(a, x)
        Leaves(u, y),
        {z ≐ x•̇ y};
```

Trees of the form

Leaves(*a*, *x*)

are admissible only if *x* is the list of leaves of the finite tree *a* (not including the leaves labeled <>α)). The query

Leaves(height("Max",<180/100,meters>,1'), *x*)?

produces the answer

{*x* = <'M', 'a', 'x', 9/5, meters, 1'>}.

Computing decimal integers

Our second example shows how approximated concatenation can be used to access the last element of a list. We want to transform a sequence of digits into the integer it represents. Here is the program without comments :

```
Value(<>, 0) → ;
Value(y, 10m+n) →
        Value(x, m),
        {y ≐ x•̇ <n>};
```

As a reply to the query

Value(<1,9,9,0>, *x*) ?

we obtain

{x = 1990}.

Computing the reverse of lists

If one knows how to access the first and the last element of a list it must be possible to write an elegant program computing the reverse of a list. Here is the one I propose :

Reverse(*x*, *y*) $\rightarrow$
    Palindrome(*u*),
    $\{u \doteq x \bullet y, |x| = |y|\}$;

Palindrome(<>) $\rightarrow$;
Palindrome(*v*) $\rightarrow$
    Palindrome(*u*),
    $\{v \doteq <a> \bullet u \bullet <a>\}$;

Each of the two queries

Reverse(<1,2,3,4,5>, *x*) ?
Reverse(*x*, <1,2,3,4,5>) ?

produces the same answer

{*x* = <5,4,3,2,1>}.

For the query

Reverse(*x*, *y*) Reverse(*y*, *z*), {*x* ? *z*, |*x*| = 10} ?

we get no answer at all, which confirms that reversing a list twice yields the initial list.

Context-free recognizer

The treatment of concatenation provides a systematic and natural means of relating "context-free" grammar rules with Prolog III rules, thus constructing a recognizer. Let us for example consider the grammar

$$\{S \rightarrow AX,\ \ A \rightarrow \wedge,\ \ A \rightarrow aA,\ \ X \rightarrow \wedge,\ \ X \rightarrow aXb\}$$

which defines the language consisting of sequences of symbols of the form $a^m b^n$ with $m = n$. The following program corresponds to the grammar :

```
Sform(u) →
        Aform(v)
        Xform(w),
        {u ≐ v• w};

Aform(u) →
        {u = <>};

Aform(u) →
        Aform(v),
        {u = "a"•v};

Xform(u) →
        {u = <>};

Xform(u) →
        Xform(v),
        {u ≐ "a"• v• "b"};
```

The query

Sform("aaabb") ?

produces the answer

{}

which signifies that the string "aaabb" belongs to the language, whereas the query

Sform("aaabbbb") ?

produces no response, which means that the string "aaabbbb" does not belong to the language.

TREATMENT OF INTEGERS

The algorithms used for solving constraints on integers are complex and quite often inefficient. It is for this reason that the structure underlying Prolog III does not contain a relation restricting a number to be only an integer. We have however considered a way of enumerating integers satisfying the set of current constraints.

Enumeration of integers

The Prolog III abstract machine is modified in such a way as to behave as if the following infinite set of rules

enum(0) →;
enum(–1) →;
enum(1) →;
enum(–2) →;
enum(2) →;
....

had been added to every program. Moreover the abstract machine is implemented in such a way as to guarantee that the search for applicable rules takes a finite among of time whenever this set is itself finite. In connection with the definition of the abstract machine this can be regarded as adding all the transitions of the form

$$(\mathrm{W}, t_0\, t_1 ... t_m\, , \mathrm{S}) \;\; \rightarrow \;\; (\mathrm{W}, t_1 ... t_m\, , \mathrm{S} \approx \{t_0 = \mathrm{enum}(\mathbf{n})\}),$$

where $\mathbf{n}$ is an integer such that the system $\mathrm{S} \approx \{p_0 = \mathrm{enum}(\mathbf{n})\}$ admits at least one solution in which the values of the t_i 's are all defined.

For example, if in the current state of the abstract machine the first term to be deleted is « enum(x) » and if the system S of constraints is equivalent on $\{x\}$ to $\{3/4 = x, x = 3+1/4\}$, then they will be two transitions : one to a state with a system equivalent to $\mathrm{S} \approx \{x=1\}$, the other to a state with a system equivalent to $\mathrm{S} \approx \{x=2\}$.

Let us add in this connection that if S is a system forcing the variable x to represent a number, then, in the most complex case, the system S is equivalent on $\{x\}$ to a system of the form

$$\{x = a_0, x\ ?\ a_1, \dots , x\ ?\ a_n, x\ = a_{n+1}\},$$

where the a_i 's are rational numbers.

A problem, taken from one of the many books of M. Gardner [15], illustrates nicely the enumeration of integers. The problem goes like this. In times when prices of farm animals were much lower than today, a farmer spent \$100 to buy 100 animals of three different kinds, cows, pigs and sheep. Each cow cost \$10, each pig \$3 and each sheep 50 cents. Assuming that he bought at least one cow, one pig and one sheep, how many of each animal did the farmer buy ?

Let x, y and z be the number of cows, pigs and sheep that the farmer bought. The query

$$\text{enum}(x)\ \text{enum}(y)\ \text{enum}(z),$$
$$\{x+y+z = 100,\ 10x+3y+z/2 = 100,\ x = 1,\ y = 1,\ z = 1\}\ ?$$

produces the answer

$$\{x = 5, y = 1, z = 94\}.$$

This problem reminds us of a problem mentioned at the beginning of this paper. Find the number x of pigeons and the number y of rabbits such that together there is a total of 12 heads and 34 legs. It was solved by putting the query

$$\{x+y = 12,\ 2x+4y = 34\}\ ?$$

But, given that, a priori, we have no guarantee that the solutions of this system are non-negative and integer numbers, it is more appropriate to put the query

$$\text{enum(x) enum(y)},\ \{x+y = 12,\ 2x+4y = 34,\ \text{x} = 0,\ \text{y} = 0\}\ ?$$

which produces the same answer

$$\{x = 7, y = 5\}.$$

Cripto-arithmetic

Here is another problem that illustrates the enumeration of integers. We are asked to solve a classical cripto-arithmetic puzzle : assign the ten digits 0,1,2,3,4,5,6,7,8,9 to the ten letters *D*,*G*,*R*,*O*,*E*,*N*,*B*,*A*,*L*,*T* in such a way that the addition *DONALD* + *GERALD* = *ROBERT* holds. We deterministically install the maximal number of constraints on the reals and use the non-determinism to enumerate all the integers which are to satisfy these constraints. Here is the program without any comments :

```
Solution(i, j, i+j) →
        Value(<D,O,N,A,L,D>, i)
        Value(<G,E,R,A,L,D>, j)
        Value(<R,O,B,E,R,T>, i+j)
        DifferentAndBetween09(x)
        Integers(x),
        {<D,G,R,E,N,B,A,L,T,O> = x,
        D ? 0, G ? 0, R ? 0};

Value(<>, 0)→ ;
Value(y, 10i+j) →
        Value(x, i), {y ≐  x• <j>};

DifferentAndBetween09(<>) →;
DifferentAndBetween09(<i>•x) →
        OutOf(i, x)
        DifferentAndBetween09(x),
        {0 = i, i = 9};

OutOf(i, <>) →;
OutOf(i, <j>•x) →
        OutOf(i, x), {i ? j};

Integers(<>) →;
Integers(<i>•x) →
        enum(i) Integers(x);
```

The answer to the query

$$\text{Solution}(i, j, k)\ ?$$

is

$$\{i = 526485,\ j = 197485,\ k = 723970\}.$$

Self-referential puzzle

The last example is a typical combinatorial problem that is given a natural solution by enumeration of integers in a involving approximated concatenation and multiplication. Given a positive integer n, we are asked to find n integers x_1, .., x_n such that the following property holds :

"In the sentence that I am presently uttering, the number 1 occurs x_1 times, the number 2 occurs x_2 times, ... , the number n occurs x_n times".

We proceeds as if one were using true (and not approximated) concatenation and one writes the program whose admissible trees are of the form

$$\text{Counting}(<x_1,...,x_m>,\ <y_1+1,...,y_n+1>),$$

each x_i being an integer between 0 and m, each y_i being the number of occurrences of the integer i in the list $<x_1,...,x_m>$. This is the program :

```
Counting(<>, Y) →,
        {<1>•Y = Y•<1>};
Counting(<x>•X, U•<y+1>•V) →
        Counting(X, U•<y>•V),
        {|U | = x–1};
```

Here the constraint $\{<1>\bullet Y = Y\bullet<1>\}$ is an elegant way of forcing Y to be a list of 1's. If everything were perfect, it would suffice to ask the query "Counting(X, X), $\{|X| = n\}$" to obtain the list of the desired n integers. Prolog III not being perfect, we have to substitute approximated concatenation for true concatenations. We must therefore complete the program with an enumeration of the integers x_1, .., x_n that we are looking for. All the lists are thus constrained to be of integer length, that is to say, to be true lists and as a result all the approximated concatenations become true concatenations. The first property is

$$x_1 + \dots + x_n = 2n,$$

which expresses that the total number of occurrences of numbers in the sentences is both $x_1+\dots+x_n$ and $2n$. The second is

$$0x_1+1x_2+ \dots +(\mathrm{n}-1)x_n = n(n+1)/2,$$

which expresses that the sum of numbers which appear in the sentences is both $1x_1+2x_2+\dots+nx_n$ and $x_1+\dots +x_n + 1+\dots+n$. From all these considerations the following final program results :

```
Solution(X) →
        Sum(X, 2n)
        WeightedSum(X, m)
        Counting(X, X)
        Integers(X),
        {n = |X|, m ≐ n∞(n+1) /2};

Sum(<>, 0) →;
Sum(<x>•X, x+y) →
        Sum(X, y);

WeightedSum(<>, 0) →;
WeightedSum(X•<x>, z+y) →
        WeightedSum(X, y),
        {z ≐ |X|∞ x};

Counting(<>, Y) →,
        {<1>•Y ≐ Y• <1>};
Counting(<x>•X, Y') →
        Counting(X, Y),
        {Y' ≐ U• <y+1>• V,
        Y ≐ U• <y>• V,
        |U | = x–1};

Integers(<>) →;
Integers(<x>•X) →
        Integers(X)
        enum(x);
```

Assigning successively to *n* the values 1,2, ... , 20 and asking the query

Solution(*X*), {|*X*| = *n*} ?

we obtain as answers

{*X* = <3,1,3,1>},
{*X* = <2,3,2,1>},
{*X* = <3,2,3,1,1>},
{*X* = <4,3,2,2,1,1,1>},

{*X* = <5,3,2,1,2,1,1,1>},
{*X* = <6,3,2,1,1,2,1,1,1>},
{*X* = <7,3,2,1,1,1,2,1,1,1>},
{*X* = <8,3,2,1,1,1,12,1,1,1>},
....
{*X* = <16,3,2,1,1,1,1,1,1,1,1,1,1,1,1,2,1,1,1>},
{*X* = <17,3,2,1,1,1,1,1,1,1,1,1,1,1,1,1,2,1,1,1>}.

The regularity in the answer gives rise to the idea of proving that for $n = 7$ there always exists a solution of the form

$$x_1, \ldots, x_n \quad = \quad n{-}3, 3, 2, 1, \ldots, 1, 2, 1, 1, 1.$$

PRACTICAL REALIZATION

Prolog III is of course more than an intellectual exercise. A prototype of a Prolog III interpreter has been running in our laboratory since the end of 1987. A commercial version based on this prototype is now being distributed by the company PrologIA at Marseilles (Prolog III version 1.0). This product incorporates the functionalities described in the present paper as well as facilities calculating maximum and minimum values of numerical expressions. We have been able to use it to test our examples and to establish the following benchmarks (on a Mac II, first model).

Light meals	4 sec
Instalments, n = 3	2 sec
Instalments, n = 50	6 sec
Instalments, n = 100	23 sec
Periodic sequence	3 sec
Squares, n = 9	13 min 15 sec
Squares, n = 9, 1st solution	1 min 21 sec
Squares, n = 10, 1st solution	6 min 36 sec
Squares, n = 11, 1st solution	1 min 38 sec
Squares, n = 12, 1st solution	5 min 02 sec
Squares, n = 13, 1st solution	4 min 17 sec
Squares, n = 14, 1st solution	13 min 05 sec
Squares, n = 15, 1st solution	11 min 29 sec
Faults detection, 2nd query	3 sec
Lewis Carrol, 2nd query	3 sec
Donald+Gerald...	68 sec
Self-referential-puzzle, n = 4	3 sec
Self-referential-puzzle, n = 5	4 sec
Self-referential-puzzle, n = 10	11 sec
Self-referential-puzzle, n = 15	36 sec
Self-referential-puzzle, n = 20	1 min 54 sec
Self-referential-puzzle, n = 25	5 min 51 sec
Self-referential-puzzle, n = 30	17 min 55 sec

All the above figures, except when stated otherwise, are the execution times of complete programs including the input and output of queries and answers. The instalment calculation consist in computing a sequence of instalments i, $2i$, $3i$, ... ,ni needed to reimburse a capital of 1000. In order to do justice to these results one must take into account the fact that all the calculations are carried out in infinite precision. In the instalment example with n=100 a simplified fraction with a numerator and a denominator with more than 100 digits is produced !

We finish this paper with information on the implementation of Prolog III. The kernel of the Prolog III interpreter consists of a two-stack machine which explores the search space of the abstract machine via backtracking. These two stacks are filled and emptied simultaneously. In the first stack one stores the structures representing the states through which one passes. In the second stack one keeps track of all the modifications made on the first stack by address-value pairs in order to make the needed restorations upon backtracking. A general system of garbage collection [23] is able to detect those structures that have become inaccessible and to recuperate the space they occupy by compacting the two stacks. During this compaction the topography of the stacks is completely retained. The kernel of the interpreter also contains the central part of the solving algorithms for the = and ? constraints. These algorithms are essentially an extension of those already used in Prolog II and described in [10]. The extension concerns the treatment of list concatenation and the treatment of linear numerical equations containing at least one variable not restricted to represent a non-negative number. A general mechanism for the delaying of constraints, which is used to implement approximated multiplication and concatenation, is also provided in the kernel. Two submodules are called upon by the interpreter, one for the treatment of Boolean algebra, the other for the remaining numerical part.

The Boolean algebra module works with clausal forms. The algorithms used [2] are an incremental version of those developed by P. Siegel [24], which are themselves based on SL-resolution [20]. On the one hand they determine if a set of Boolean constraint is solvable and on the other they simplify these constraints into a set of constraints containing only a minimal subset of variables. Related experiments have been performed with an algorithm based on model enumeration [21]. Although significant improvement has been achieved as far as solvability tests are concerned a large part of these ameliorations is lost when it comes to simplifying the constraints on output. Let us mention that W. Büttner and H. Simonis approach the incremental solving of Boolean constraints with quite different algorithms [6].

The numerical module treats linear equations the variables of which are constrained to represent non-negative numbers. (These variables x are introduced to replace constraints of the form $p = 0$ by the constraints $x = p$ and $x = 0$). The module consists essentially in an incremental implementation of G. Dantzig's simplex algorithm [12]. The choice of pivots follows a method proposed in M. Balinski et R. Gomory [1] which, like the well-known method of R. Bland [3], avoids cycles. The simplex algorithm is used both to verify if the numerical constraints have solutions and to detect those variables having only one possible value. This allows to simplify the constraints by detecting the hidden equations in the original constraints. For example the hidden equation $x = y$ will be detected in $\{x = y, y = x\}$. The module also contains various subprograms needed for addition and multiplication operations in infinite precision, that is to say, on fractions whose numerators and denominators are unbounded integers. Unfortunately we have not included algorithms for the systematic elimination of useless numerical variables in the solved systems of constraints. Let us mention in this connection the work of J-L. Imbert [17].

ACKNOWLEDGEMENTS

The prototype interpreter was built in cooperation between our laboratory (GIA) and the company PrologIA. Substantial financial support was obtained on the one hand from the Centre National d'Etudes des Télécommunications (contract 86 1B 027) and on the other from the CEE within the framework of the Esprit project P1106 "Further development of Prolog and its Validation by KBS in Technical Areas". Additional support was granted by the Commissariat à l'Energie Atomique in connection with the Association Méditérannéenne pour le Développement de l'IA, by the DEC company in connection with an External Research Grant and by the Ministère de la Recherche et de l'Enseignement Supérieur within the two "Programmes de Recherches Concertés", "Génie Logiciel" and "Intelligence Artificielle". Finally the most recent work on approximated multiplication and concatenation has been supported by the CEE Basic Research initiative in the context of the "Computational Logic" project.

I thank the entire research team which has been working on the Prolog III interpreter : Jean-Marc Boï and Frédéric Benhamou for the Boolean algebra module, Pascal Bouvier for the supervisor, Michel Henrion for the numerical module, Touraïvane for the kernel of the interpreter and for his work on approximated multiplication and concatenation. I also thank Jacques Cohen of Brandeis University whose strong interest has been responsible for my writing this paper and Franz Guenthner of the University of Tübingen who helped in the preparation of the final version of this paper. Finally I thank Rüdiger Loos of the University of Tübingen who pointed my attention to two particularly interesting numerical problems : the periodical sequence and the filling of a rectangle by squares.

REFERENCES

1. BALINSKI M. L. and R. E. GOMORY, A Mutual Primal-Dual Simplex Method, *Recent Advances in Mathematical Programming*, McGraw-Hill, pp. 17-26, 1963.
2. BENHAMOU F. and J-M. BOÏ, *Le traitement des contraintes Booléennes dans Prolog III*, Thèses de Doctorat, GIA, Faculté des Sciences de Luminy, Université Aix-Marseille II, Novembre 1988.
3. BLAND R. G., New finite pivoting for the simplex method, *Mathematics of Operations Research*, Vol. 2, No. 2, May 1977.
4. BOOLE G., *The Laws of Thought,* Dover Publication Inc., 1958.
5. BROWN M., Problem proposed in : *The American Mathematical Monthly*, vol. 90, no. 8, pp. 569, 1983.
6. BÜTTNER W. and H. SIMONIS, Embedding Boolean Expressions into Logic Programming, *Journal of Symbolic Computation*, 4, 1987.
7. CARROLL L., *Symbolic Logic and the Game of Logic*, New York, Dover, 1958.

8. COLMERAUER A., Theoretical model of Prolog II, *Logic programming and its application*, ed. by M. Van Caneghem and D. Warren, Ablex Publishing Corporation, pp. 3-31, 1986.

9. COLMERAUER A., Prolog in 10 figures, *Communications of the ACM,* Volume 28, Number 12, pp. 1296-1310, December 1985.

10. COLMERAUER A., Equations and Inequations on Finite and Infinite Trees, Invited lecture, *Proceedings of the International Conference on Fifth Generation Computer Systems,* Tokyo, pp. 85-99, November 1984.

11. COLMERAUER A., *Final Specifications for Prolog III*, Esprit P11O6 report : Further development of Prolog and its Validation by KBS in Technical Areas, February, 1988.

12. DANTZIG G. B., *Linear Programming and Extensions*, Princeton University Press, 1963.

13. DINCBAS M. ET AL., The Constraint Logic Programming CHIP, *Proceedings of the International Conference on Fifth Generation Computer Systems,* ICOT, pp. 693-702, 1988.

14. DUIJVESTIJN A. J. W., Simple Perfect Squared Square of Lowest Order, *Journal of Combinatorial Theory,* Series B 25, pp. 240-243, 1978.

15. GARDNER M., Wheels, life and other mathematical amusements, W. H. Freeman and Compagny, 1983.

16. GENESERETH M. R. and M. L. GINSBERG, Logic Programming, *Communications of the ACM,* Volume 28, Number 9, 933-941, September 1985.

17. IMBERT J-L., About Redundant Inequalities Generated by Fourier's Algorithm, AIMSA'90, *4th International Conference on Artificial Intelligence : Methodology, Systems, Applications*, Albena-Varna, Bulgaria, September 1990, Proceedings to be published by North-Holland,

18. JAFFAR J. and J-L. LASSEZ, Constraint Logic Programming, *14th ACM Symposium on the Principle of Programming languages*, pp. 111-119, 1987.

19. JAFFAR J. and S. MICHAYLOV, Methodology and Implementation of a Constraint Logic Programming System, *Proceedings of the Fourth International Conference on Logic Programming,* Melbourne, MIT Press, pp. 196-218, 1987.

20. KOWALSKI R. and D. KUEHNER, Resolution with Selection Function, *Artificial Intelligence*, Vol. 3, No. 3, pp. 227-260, 1970.

21. OXUSOFF L. and A. RAUZY., *Evaluation sémantique en calcul propositionnel*, Thèses de Doctorat, GIA, Faculté des Sciences de Luminy, Université Aix-Marseille II, January 1989.

22. ROBINSON A., A machine-oriented logic based on the resolution principle, *Journal of the ACM*, 12, December 1965.

23. TOURAÏVANE, *La récupération de mémoire dans les machines non déterministes*, Thèse de Doctorat, Faculté des Sciences de Luminy, Université Aix-Marseille II, November 1988.

24. SIEGEL P., *Représentation et utilisation de la connaissance en calcul propositionnel*, Thèse de Doctorat d'Etat, GIA, Faculté des Sciences de Luminy, Université Aix-Marseille II, July 1987.

25. SPRAGUE R., Über die Zerlegung von Rechtecken in lauter verschiedene Quadrate, *Journal für die reine und angewandte Mathematik*, 182, 1940.