PROLOG AND INFINITE TREES

Alain Colmerauer

Groupe Intelligence Artificielle
Faculté des Sciences de Luminy
Université Aix-Marseille II
13288 Marseille cedex 2, France

ABSTRACT

The paper deals with the manipulation of infinite trees in the context of the programming language Prolog. With this purpose a novel and concise model of Prolog is presented. The model does not explicity involve the first order logic. The problem of unifying two terms is replaced by that of determining whether or not a system of equations has at least one solution. Several examples of the use of infinite trees are also given.

1. FOREWORD

Prolog is a programming language that has been developed in Marseille, mainly by Roussel and Colmerauer in (Battani, 1973; Colmerauer et al. 1973; Roussel, 1975). The name "Prolog" was given by Roussel for "Pro(gramming) in log(ic)". Originally, it was a theorem prover based on the resolution principle of Robinson (1965), with strong restrictions to reduce the search space: linear proof, unification applying only on the first literal of each clause, etc... Credit is given to van Emden and Kowalski (1976) for having proposed a very simple theoretical model, the Horn clauses, which explains our restrictions and specifies exactly what Prolog tries to compute (minimal Herbrand interpretation).

However, there is a basic difference between most Prolog interpreters and the theoretical model: for efficiency purposes the interpreters utilize a simplified variant of the unification algorithm. The simplification consists of

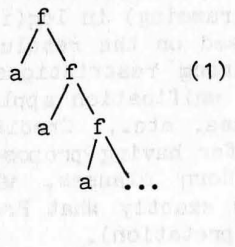
suppressing the "occur check", that is, to allow the unification of a variable with a term already containing this variable. This simplification may be unsafe but the efficiency gains are considerable. For example, with the "occur check", the concatenation of two lists requires (at least) a time proportional to the square of the size of the first list. If the "occur check" is eliminated the time becomes linear. This is the case even when considering the fastest unification algorithms available (Baxter, 1976; Martelli and Montanari, 1976; Paterson and Wegman, 1976).

The objective of this paper is to describe a novel theoretical model of Prolog involving infinite trees. This model corresponds to the Prolog interpreters which do not perform the "occur check". However, the unification algorithm for these interpreters must be modified so as to assure termination. The best way to achieve this termination is an open problem which is not discussed here.

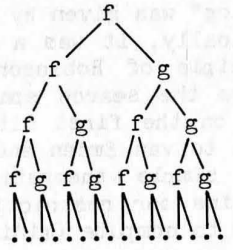
The availability of elaborate data structures such as as infinite trees allow novel solutions to programming problems. Several examples of these solutions are given. The examples were tested using a recent Prolog system developed by Colmerauer et al. (1981) and which incorporates the ideas proposed herein.

2. RATIONAL TREES

In Prolog each variable represents a finite tree which is constructed over a set F of functional symbols. To unify the variable x with the term f(a,x), means that x is also allowed to represent the infinite tree in (1).



(1)

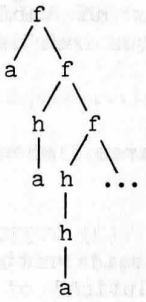


(2)

An infinite tree is a tree with an infinite set of nodes. We need only consider a special class of infinite trees: the "rational" trees.

DEFINITION: a "rational" tree is a tree which has a finite set of subtrees.

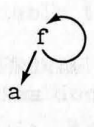
The tree in (1) is rational because it contains only the two subtrees: itself and the one node tree a. The tree in (2) is also rational, because it contains only two subtrees. However, the tree:



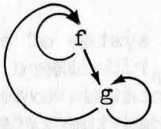
(3)

is not rational since the set of its subtrees is not finite.

A rational tree may be transformed into a finite diagram by "merging" all the nodes from which the same subtrees start. For the two previous rational trees we obtain respectively:

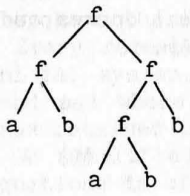


(4)



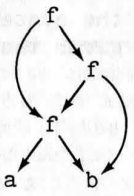
(5)

These diagrams are the "minimal representations" of the two infinite trees. The number of nodes in each diagram equals the number of subtrees of the tree. Notice that even the minimal representation of a finite tree may contain fewer nodes than the tree itself. For example:



becomes

(6)



(7)

3. SYSTEM OF EQUATIONS

In addition to the set F of functional symbols, we introduce

an infinite set V of variables which allows us to construct terms over $F_U V$.

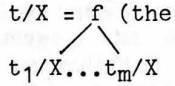
A "tree-assignment" X is a finite set of ordered pairs of the form:

$$X = \{x_1:=r_1,\ x_2:=r_2,\ \ldots,\ x_n:=r_n\}$$

where the r_i's are (possibly infinite) trees and the x_i's are distinct variables. If a given term t contains no variables other than the x_i's in X, then t/X denotes the tree defined by:

if $t = x_i$ then $t/X = r_i$

if $t = ft_1 \ldots t_m$ then $t/X =$ f (the one node tree f when m=o)



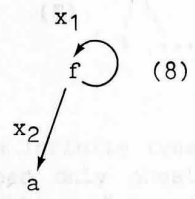
An "equation" is an ordered pair of terms (s,t) written as s=t and a tree-assignment X is a "tree-solution" of this equation iff s/X= t/X.

By systems of equations we mean only finite sets of equations. A tree-assignment X is a tree-solution of a system E of equations iff X is a subset of a tree-assignement which is a tree-solution of every equation in E.

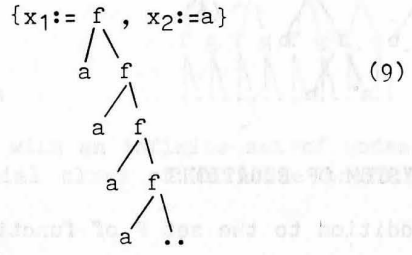
Let us consider a system of equations of the form: $\{x_1=u_1,\ \ldots,\ x_n=u_n\}$, where the x_i's are distinct variables and each u_i contains exactly one functional symbol and no variables other than the x_i's.

It can be shown that such a system has exactly one tree-solution of the form $\{x_1:=r_1,\ \ldots,\ x_n:=r_n\}$ and that the r_i's are rational trees. The proof of this property is beyond the scope of this paper.

For example, to the system: $\{x_1=f(x_2,x_1),\ x_2=a\}$ corresponds diagram (8) which gives the tree-solution in (9).



(8)



(9)

This leads to the first result:

SOLVABLE FORM: A system of equations of the following form has at least one tree-solution: $\{x_1=t_1,\ \ldots,x_n=t_n\}$, where the x_i's are distinct variables and the t_i's are any terms.

The second result is obvious:

UNSOLVABLE FORM: A system of equations containing an equation of the following form has no tree-solution:

$fs_1 \ldots s_n = gt_1 \ldots t_m$, where f,g are distinct functional symbols.

We now introduce five transformations on systems of equations:

COMPACTION (C): Eliminate any equation of the form x=x where x is a variable.

VARIABLE ELIMINATION (VE): If x,y are distinct variables, x=y is in the system and x has other occurrences in that system then replace these other occurrences of x by occurrences of y. (x=y stays)

VARIABLE ANTEPOSITION (VA): If x is a variable and t is not a variable then replace t=x by x=t.

CONFRONTATION (CO): If x is a variable and t_1, t_2 are not variables and $|t_1| \leq |t_2|$ then replace $\{x=t_1, x=t_2\}$ by $x=t_1, t_1=t_2$. By $|t|$ we denote the size of t, i.e. the number of occurrences of elements of $F_U V$.

SPLITTING (S): Replace $\{fs_1..s_1=ft_1..t_n\}$ by $\{s_1=t_1,..,s_n=t_n\}$

These transformations have interesting properties:
- They preserve equivalence. Two systems are equivalent iff they have the same tree-solutions.
- Every repeated application of these transformations, on an initial system, leads, after a finite number of steps, to a dead end where no transformation can be applied. The system thus obtained is a "simplified form" of the initial system.
- A simplified form has a tree-solution iff each of its equations is of the form x=t, where x is a variable.

The first property is obvious, the second will be proved at the end of this chapter and the third is a direct consequence of our solvable and unsolvable forms.

For example, let x,y be variables, h a unary functional symbol and k a 0-ary functional symbol.

The system: {x=hhx, hhk=x} has no tree-solution because of the transformations in (10).

The system: {x=y, x=hhx, y=hhhy} has a tree-solution because of the transformations in (11).

```
{x=hhx, hhk=x}     (VA)          {x=y, x=hhx, y=hhhy}     (VE)
{x=hhx, x=hhk}     (CO)          {x=y, y=hhy, y=hhhy}     (CO)
{x=hhx, hhx=hhk}   (S)   (10)    {x=y, y=hhy, hhy=hhhy}   (S)   (11)
{x=hhx, hx=hk}     (S)           {x=y, y=hhy, hy=hhy}     (S)
{x=hhx, x=k}       (C)           {x=y, y=hhy, y=hy}       (CO)
{k=hhx, x=k}                     {x=y, hy=hhy, y=hy}      (S)
                                 {x=y, y=hy}
```

Note that the test in the confrontation is necessary: The following loop results from eliminating the test in the second confrontation of the last example:

```
{x=y, y=hhy, y=hy}    (CO, without test)
{x=y, y=hhy, hhy=hy}  (S)                    (12)
{x=y, y=hhy, hy=y}    (VA)
{x=y, y=hhy, y=hy}    (CO, without test)
```

By introducing intermediate variables so that every term contains no more than one functional symbol the check for a greatest term is no longer necessary. The entire simplification process becomes similar to the efficient unification algorithm on general trees proposed by Huet (1976). The intermediate variables correspond to the pointers needed to represent a term in a computer. In our Prolog implementation we avoid the test in another way. We order the equations and then apply the transformations under certain conditions depending on the ordering. Unfortunately we have not been able to obtain a proof of the termination of this algorithm, but it has never failed in any of the programs we have run.

We now give the outline of the proof that it is impossible to construct an infinite sequence of systems E_i of equations:

$$E_0, E_1, \ldots, E_i, E_{i+1}, \ldots$$

where E_{i+1} is obtained by applying one of our five transformations to E_i.

First we prove that such a sequence has to be finite if we do not apply the splitting transformation. Let us consider the initial system E_0. Let N be the number of equations of the form x=x or x=y and let M be the number of equations of the form x=t or t=x. The number of compactions and the number of variable eliminations cannot be greater than N. The number of variable antepositions and the number of confrontations cannot be greater than M.

Secondly, we consider the sequence of positive integers:

defined by: $\|E_0\|$, $\|E_1\|$, ..., $\|E_i\|$, $\|E_{i+1}\|$, ...

$$\|\{s_1=t_1, \ldots, s_n=t_n\}\| = {}_k\max\{|s_1|, |t_1|\} + \ldots + {}_k\max\{|s_n|, |t_n|\}$$

where k is the greatest n-arity of functional symbols occurring in E_0. From the definition of the transformations, it is possible to show that:

$\|E_i\| \geq \|E_{i+1}\|$ for all the transformations, and
$\|E_i\| > \|E_{i+1}\|$ in the case of a splitting transformation.

Our first result proved that the existence of an infinite sequence of E_i's implies an infinite number of applications of the splitting transformation. From the above inequalities it follows that this infinite sequence is impossible because the integers $\|E_i\|$ would become negative.

4. RECURSIVE DEFINITION OF ASSERTIONS

Let F be a set of functional symbols and let V be an infinite enumerable set of variables. A Prolog program is basically a recursive definition of a subset A of the (possibly infinite) trees over F. The elements of this subset are called "assertions". This recursive definition has to be elaborated by means of set of rule schemas, each being of the form:

$t \rightarrow t_1 \ldots t_n$ in which n can be equal to 0 and $t, t_1, \ldots, t_n$ are terms over $F \cup V$.

These rule schemas induce a set of rules: $t/X \rightarrow t_1/X \ldots t_n/X$ obtained by considering all the possible tree-assignments X to the variables.

Each of these rules: $r \rightarrow r_1 \ldots r_n$
can be interpreted in two different ways:

1. As the "rewriting rule":
   r can be rewritten in the sequence $r_1 \ldots r_n$
   and thus, when n=o, as: r can be erased.

2. As the "logical implication" dealing with the subset A of trees:
   r_1 element of A and ... and r_n element of A implies r element of A.
   For n=0, this becomes: r element of A.

Depending on whether we use the first or the second interpretation, the "assertions" are defined by:

DEFINITION 1: The "assertions" are the trees which can be erased in a finite number of steps, using the "rewriting rules".

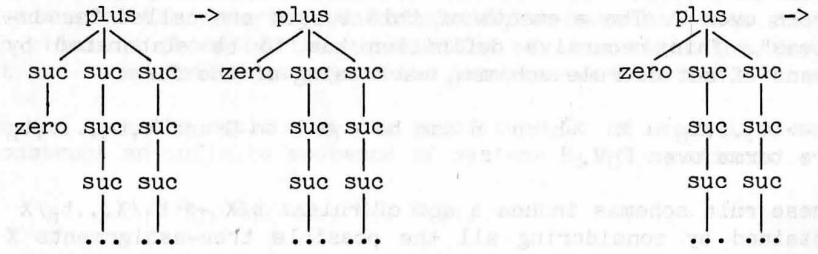
DEFINITION 2: The "assertions" are the smallest subset A of trees which satisfy the "logical implications".

It is shown in (Colmerauer, 1979b) that the two definitions are equivalent. The second has the advantage of being more abstract and related to logic. The first definition is more akin to the way the Prolog interpreter works.

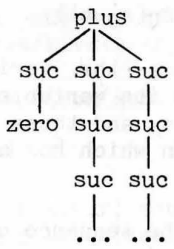
Consider for example the well known Prolog program consisting of the two schemas:

```
plus(suc(x),y,suc(z)) -> plus(x,y,z)
plus(zero,x,x) ->
```

in which x,y,z are variables and zero,suc,plus are functional symbols of n-arity 0,1,3 respectively. These schemas induce, among others, the two rules:



According to the first definition the rational tree:



is an assertion, because it can be erased in two steps using the two preceding rules as rewriting rules. According to the second definition the above tree is also an assertion, because any set E which satisfies the logical implications of these two rules must contain this tree. Note that this tree being an assertion can be viewed as the existence of an x such that 1+x=x. If this is to be avoided one should give a more precise definition of plus(x,y,z), that is:

```
plus(suc(x),y,suc(z)) -> plus(x,y,z)
plus(zero,x,x) -> integer(x)

integer(zero) ->
integer(suc(x)) -> integer(x)
```

5. COMPUTATION OF SUBSETS OF ASSERTIONS

We have just explained what a Prolog program means, but not what it does. When the Prolog interpreter is invoked, it tries to solve the following problem:
- given a program, which is a recursive definition of a set of assertions A,
- given a set of terms $\{t_1,t_2,\ldots,t_n\}$ which acts like a "window" over the assertions, and given the set of variables $\{x_1,\ldots,x_m\}$ which occur in it,
- find all the assertions which can be "seen" through this window, i.e.: compute all the tree-assignments of the form $X = \{x_1:=r_1,\ldots,x_m:=r_m\}$ such that $\{t_1/X,\ldots,t_n/X\}$ becomes a subset of A.

To do so we will work on pairs of the form:

$(t_1 \ldots t_n, E)$ where $t_1 \ldots t_n$ is a (possibly empty) sequence of terms, and E a system of equations.

DEFINITION OF "=>": We introduce the following binary relation between the above pairs:

$(s_0 s_1 \ldots s_m, E) \Rightarrow (t_1 \ldots t_n s_1 \ldots s_m, E \cup \{s_0 = t\})$ iff

$t \rightarrow t_1 \ldots t_n$ is obtained by renaming the variables of a schema in such a way that its variables are disjoint from those of $s_0 \ldots s_m$ and those of E, and

$E \cup \{s_0 = t\}$ is a system of equation which has at least one tree-solution.

We write $x \Rightarrow^* y$ iff there exist a finite sequence of u_i's such that $x = u_0$, $u_0 \Rightarrow u_1$, $u_1 \Rightarrow u_2, \ldots u_n = y$.

The following is the main result on which the Prolog interpreter is based:

WINDOW PRINCIPLE: let $\{t_1, \ldots, t_n\}$ be a set of terms, let $\{x_1, \ldots, x_m\}$ be the set of its variables, and let X be a tree-assignement of the form $X = \{x_1 := r_1, \ldots, x_m := r_m\}$:

$\{t_1/X_1, \ldots, t_n/X\}$ is a subset of the set A of assertions iff X is a tree-solution of a system E of equations such that: $(t_1 \ldots t_n, \{\}) \Rightarrow^* (\text{empty}, E)$

The proof of a similar result is found in (Colmerauer, 1979b). A more precise explanation of the Prolog interpreter is now given. Let S_0 be the sequence of terms in the window and E_0 the empty system. The interpreter enumerates all the sequences of pairs (S_i, E_i) which are such that:

$$(S_0, E_0) \Rightarrow (S_1, E_1) \Rightarrow (S_2, E_2) \Rightarrow \ldots$$

The interpreter expects all these sequences to be finite (It is up to the programmer to assure that this is the case). The ability of printing, at any time, a tree satisfying the current set of equations E_i, is one of the multiple features which have to be added to the theoretical model of Prolog to render it usable as a programming language.

Let us consider again the rules schemas:

```
plus(zero,w,w) ->
plus(suc(x),y,suc(z)) -> plus(x,y,z)
```

Assume that we wish to compute all the tree-assignements:

$X = \{u := r_1, v := r_2\}$ such that the set:

$\{plus(suc(zero),u,v)/X, plus(suc(zero),v,u))/X\}$

becomes a subset of the assertions. Then the pairs (S_i, E_i) become successively (E_i' is a simplified form of E_i):

S_0 = plus(suc(zero),u,v) plus(suc(zero),v,u)
E_0 = {}
E_0' = {}

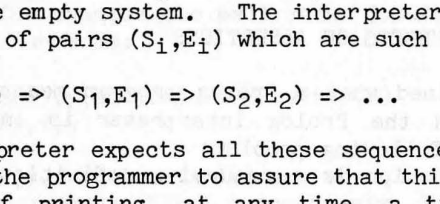
S_1 = plus(x,y,z) plus(suc(zero),v,u)
E_1 = $E_0 \cup$ {plus(suc(zero),u,v)=plus(suc(x),y,suc(z))}
E_1' = {u=y,v=suc(z),x=zero}

S_2 = plus(suc(zero),v,u)
E_2 = $E_1 \cup$ {plus(x,y,z)=plus(zero,w,w)}
E_2' = {u=w=y=z,v=suc(z),x=zero}

S_3 = plus(x',y',z')
E_3 = $E_2 \cup$ {plus(suc(zero),v,u)=plus(suc(x'),y',suc(z'))}
E_3 = {u=w=y=z=suc(z'),v=y'=suc(z),x=zero,x'=zero}

S_4 = empty
E_4 = $E_3 \cup$ {plus(x',y',z')=plus(zero,w',w')}
E_4' = {u=w=y=z=suc(z'),v=w'=y'=z'=suc(z),x=zero,x'=zero}

It follows that the only solution is:



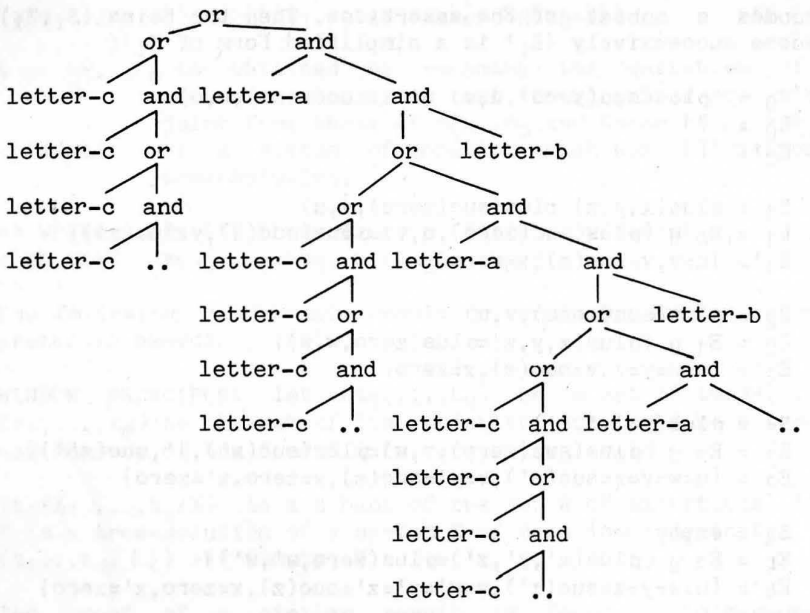
6. EXAMPLE USE OF INFINITE TREES

6.1 Grammars

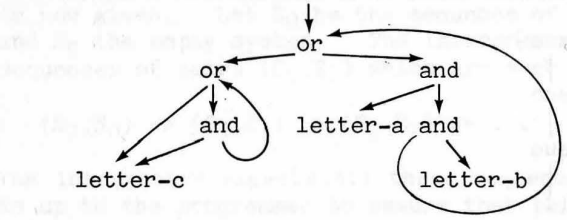
Consider the following context-free grammar:

```
S -> N        non-terminals: {S,N }
S -> aSb      initial non-terminal: S
N -> c
N -> cN       terminals: {a,b,c}
```

If we trace the rules of this grammar, starting from S, we obtain the infinite and-or tree which enumerates the generated strings ("and" is to be understood as "followed-by"):



The above is a rational tree whose minimal representation is:



Let us assume that variables are represented by one-letter identifiers. The program to define the grammar is:

```
grammar(s) ->
   equal(s,or(n,and(letter-a,and(s,letter-b))))
   equal(n,or(letter-c,and(letter-c,n)))

equal(x,x) ->
```

Terminal strings are defined by:

```
terminal-string(nil) ->
terminal-string(and(a,x)) -> terminal(a) terminal-string(x)

terminal(letter-a) ->
terminal(letter-b) ->
terminal(letter-c) ->
```


Examples of terminal strings are:

```
string-1(and(letter-a,and(letter-c,and(letter-b,nil)))) ->
string-2(and(letter-a,and(letter-b,and(letter-b,nil)))) ->
```

Consider the term produces(p,x) with:

produces(p,x) becomes an assertion	iff	p becomes a grammar and x becomes a terminal string and x can be generated by p.

It can be defined by:

```
produces(p,x) -> gives(p,x,nil)

gives(or(p,q),x,y) -> gives(p,x,y)
gives(or(p,q),x,y) -> gives(q,x,y)
gives(and(p,q),x,z) -> gives(p,x,y) gives(q,y,z)
gives(a,and(a,x),x) -> terminal(a)
```

The reader will have guessed that:

gives(p,x,y) becomes an assertion	iff	p becomes a grammar and x becomes a tree of the form: and(a_1,and(a_2,...and(a_n,y)...)) and $a_1a_2...a_n$ can be generated by p.

To check if the strings acb and abb belong to the considered language, we start (respectively) from:

```
grammar(p) string-1(x) produces(p,x)
grammar(p) string-2(x) produces(p,x)
```

It is also possible to use produce(p,x) to enumerate (indefinitely) all the strings x of the language defined by p. It is even possible, under certain conditions, to enumerate all the grammars p which generate x! Similary we can program produces-not(p,x) which means:

produces-not(p,x) becomes an assertion	iff	p becomes a grammar and x becomes a terminal string and x cannot be generated by p.

The program is:

```
produces-not(p,x) -> terminal-string(x) gives-not(p,x,nil)

gives-not(or(p,q),x,y) -> gives-not(p,x,y) gives-not(q,x,y)
gives-not(and(p,q),x,z) -> each-cut-fails(and(p,q),x,x,z)
gives-not(a,x,x) -> terminal(a)
gives-not(a,and(b,x),x) -> different-terminals(a,b)
gives-not(a,and(b,x),y) -> terminal(a) ends(y,x)

each-cut-fails(p,x,and(a,z),z) ->
each-cut-fails(p,x,and(a,y),z) ->
   ends(z,y) cut-fails(p,x,y,z) each-cut-fails(p,x,y,z)

cut-fails(and(p,q),x,y,z) -> gives-not(p,x,y)
cut-fails(and(p,q),x,y,z) -> gives-not(q,y,z)

ends(x,and(a,x)) ->
ends(x,and(a,y)) -> ends(x,y)
```

with of course, in this case:

```
different-terminals(letter-a,letter-b) ->
different-terminals(letter-a,letter-c) ->
different-terminals(letter-b,letter-a) ->
different-terminals(letter-b,letter-c) ->
different-terminals(letter-c,letter-a) ->
different-terminals(letter-c,letter-b) ->
```

6.2. Back to the Finite

Let us now introduce some general programs to manipulate rational trees. A rational tree has a finite set of subtrees, so it is possible to compute their list. Let us define:

subtrees(p,x) becomes an assertion iff p becomes the tree r_1, $r_1,r_2,\ldots,r_n$ are the subtrees of r_1, x becomes a tree of the form:

```
list(is(n,r_n),
        :
list(is(2,r_2),
list(is(1,r_1),nil))...)
```

We have:

```
subtrees(p,x) -> union-subtrees(nil,p,x)

union-subtrees(x,p,x) -> in(is(n,p),x)
union-subtrees(x,p,y) ->
 dominates(p,u) new-integer(x,n) unions(list(is(n,p),x),u,y)

unions(x,nil,x) ->
unions(x,list(p,u),z) -> union-subtrees(x,p,y) unions(y,u,z)
```



```
in(a,list(a,x)) ->
in(a,list(b,x)) -> in(a,x)

new-integer(nil,n) -> first-integer(n)
new-integer(list(is(n,p),x),m) -> next-integer(n,m)
```

where, for the previous grammar trees:

```
dominates(a,nil) -> terminal(a)
dominates(or(p,q),list(p,list(q,nil))) ->
dominates(and(p,q),list(p,list(q,nil))) ->
```

We must also introduce:

```
first-integer(1) ->

next-integer(1,2) ->
next-integer(2,3) ->
      :
```

From the subtrees of a tree we can compute a system of equations which define it. Consider:

equations(p,s) becomes an assertion iff p becomes a tree with $r_1,\ldots,r_n$ as subtrees, s becomes a system having $\{x_1:=r_1,\ldots,x_n:=r_n\}$ as solution the variables $x_1,\ldots,x_n$ are represented by $1,\ldots,n$, s is represented by the tree:

```
list(equal(n,f_n(i_n1,i_n2,...,i_nk_n)),
                  :
list(equal(2,f_2(i_21,i_22,...,i_2k_2)),
list(equal(1,f_1(i_11,i_12,...,i_1k_1)),
                                  nil))...)
```

The program is:

```
equations(p,s) -> subtrees(p,x) into(x,s,x)

into(nil,nil,x) ->
into(list(is(n,p),x),list(equal(n,q),s),y) ->
   similar(p,q) dominates(p,u) dominates(q,v)
   into-bis(u,v,y) into(x,s,y)

into-bis(nil,nil,x) ->
into-bis(list(p,u),list(n,v),x) ->
                    in(is(n,p),x) into-bis(u,v,x)
```

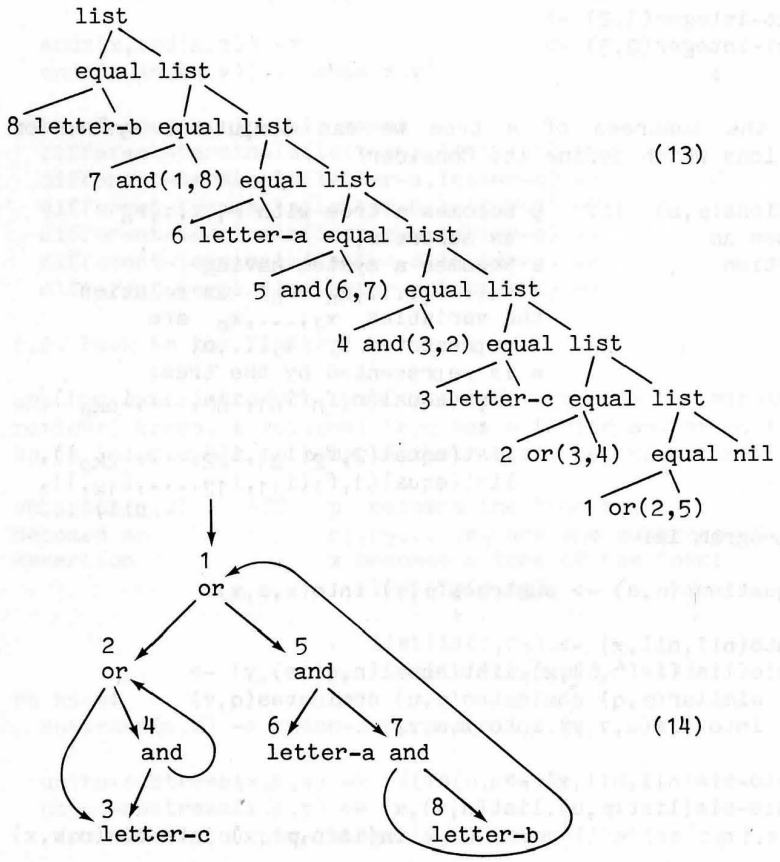
with (in the case of our grammar tree):

```
similar(a,a) -> terminal(a)
similar(or(p,q),or(x,y)) ->
similar(and(p,q),and(x,y)) ->
```

This program allows us to compute the equations which correspond to the minimal representation of a rational tree. For example, if we consider the previous grammar and we start from:

```
grammar(p) equations(p,x)
```

x becomes as first solution as shown in (13), which gives the already mentioned minimal representation of the grammar as shown in (14).



(13)

(14)

The program which defines equations(p,x) is a fundamental one. One can also use it to output infinite trees in a finite

way. For this purpose there is a simpler program which transforms any rational tree p into a finite pseudo-term q, which, in case of infinity, contains pointers to upper level nodes.

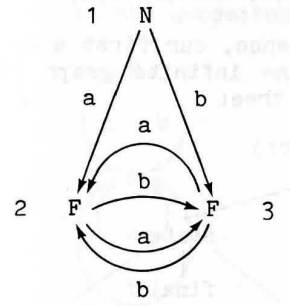
```
into-pseudo-term(p,q) -> changes(p,q,nil)

changes(p,to(n),x) -> in(is(n,p),x)
changes(p,q,x) ->
   similar(p,q) dominates(p,u) dominates(q,v)
   new-integer(x,n) change(u,v,list(is(n,p),x))

change(nil,nil,x) ->
change(list(p,u),list(q,v),x) -> changes(p,q,x) change(u,v,x)
```

If one runs the program starting from:

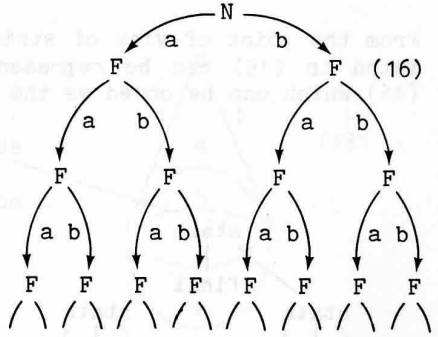
```
grammar(p) into-pseudo-term(p,q)
```

then q becomes:

6.3. Finite Automatons



(15)



(16)

A finite state automaton consists of:
- An input vocabulary. We will limit ourselves to {a,b}.
- A non-empty finite set of states which will be the nodes of a graph. There are two kinds of states: final and non-final

states (respectively labeled by F and N).
- A selected state called the initial state.
- A set of arrows connecting the states. Each arrow is labeled by an input symbol. We will only consider automata which are complete and deterministic, that is: for each node and for each input symbol there is exactly one arrow which exits the node and which is labeled by the symbol.

Our first automaton in (15) can be programmed by:

```
automaton-1(s1) -> non-final-state(s1)
                   final-state(s2)
                   final-state(s3)
                   arrow(s1,letter-a,s2) arrow(s1,letter-b,s3)
                   arrow(s2,letter-a,s3) arrow(s2,letter-b,s3)
                   arrow(s3,letter-a,s2) arrow(s3,letter-b,s2)
```

where final, non-final, and arrow will be specified later.

An automaton accepts (otherwise rejects) a string of input symbols a_1 a_2 ... an iff there exists a path of the form

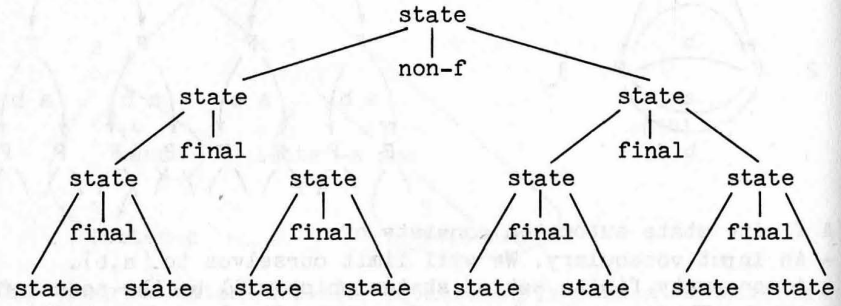
$$--a_1--> --a_2--> \ldots --a_n-->$$

which starts from the initial state and ends on a final (otherwise not final) state.

Acceptance can be programmed as:

```
accepts(s,nil,true) -> final-state(s)
accepts(s,nil,false) -> non-final-state(s)
accepts(r,list(a,x),v) -> arrow(r,a,s) accepts(s,x,v)
```

From the point of view of string acceptance, our first automaton in (15) can be represented by the infinite graph in (16) which can be coded as the rational tree:



We can now define:


```
final-state(state(x,final,y)) ->

non-final-state(state(x,non-f,y)) ->

arrow(state(x,f,y),letter-a,x) ->
arrow(state(x,f,y),letter-b,y) ->
```

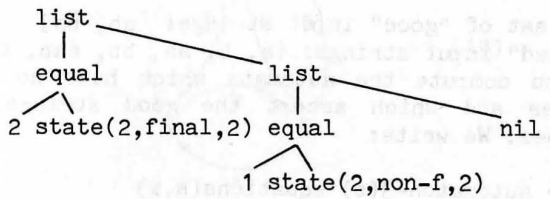
Consider the problem: compute the automaton which has the smallest number of states and which, from the point of view of acceptance, is equivalent to a given automaton. The solution to the problem is to compute the minimal representation of the rational tree which corresponds to the automaton. As we have already seen the program which defines equation(p,s), does this task. To properly execute it we need, however, to add the two rules:

```
dominates(state(x,f,y),list(x,list(y,nil))) ->
similar(state(p,f,q),state(x,f,y)) ->
```

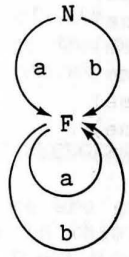
If we invoke the program, starting from :

```
automaton-1(s) equations(s,x)
```

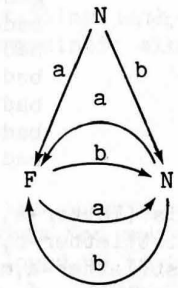
x becomes as first solution:



that is the automaton in (17).



(17)



(18)

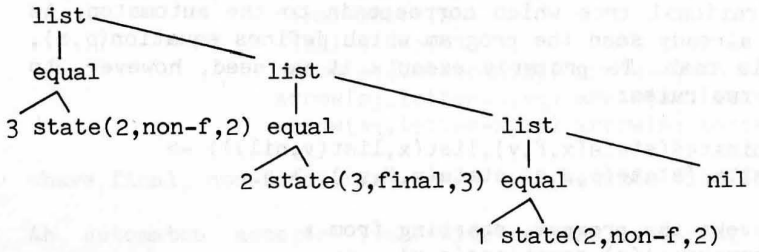
Therefore our first automaton in (15) was not minimal. Consider the second automaton in (18).

```
automaton-2(s1) -> non-final-state(s1)
                   final-state(s2)
                   non-final-state(s3)
                   arrow(s1,letter-a,s2) arrow(s1,letter-b,s3)
                   arrow(s2,letter-a,s3) arrow(s2,letter-b,s3)
                   arrow(s3,letter-a,s2) arrow(s3,letter-b,s2)
```

By starting from:

```
automaton-2(s) equations(s,x)
```

x becomes now:



which proves that the second automaton is already minimal.

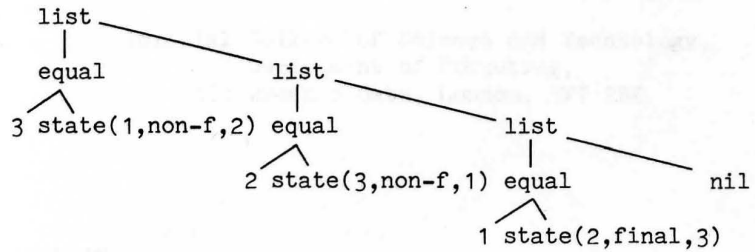
Consider now the set of "good" input strings: {ab, ba} and the set of "bad" input strings: {a, b, aa, bb, aab, bba}. The problem is to compute the automata which have no more than three states and which accept the good strings and reject the bad ones. We write:

```
solution(x) -> automaton-3(s) equations(s,x)

automaton-3(s) -> good-1(x1) accepts(s,x1,true)
                  good-2(x2) accepts(s,x2,true)
                  bad-1(y1) accepts(s,y1,false)
                  bad-2(y2) accepts(s,y2,false)
                  bad-3(y3) accepts(s,y3,false)
                  bad-4(y4) accepts(s,y4,false)
                  bad-5(y5) accepts(s,y5,false)
                  bad-6(y6) accepts(s,y6,false)

good-1(list(letter-a,list(letter-b,nil))) ->
good-2(list(letter-b,list(letter-a,nil))) ->
bad-1(list(letter-a,nil)) ->
bad-2(list(letter-b,nil)) ->
bad-3(list(letter-a,list(letter-a,nil))) ->
bad-4(list(letter-b,list(letter-b,nil))) ->
bad-5(list(letter-a,list(letter-a,list(letter-b,nil)))) ->
bad-6(list(letter-b,list(letter-b,list(letter-a,nil)))) ->
```

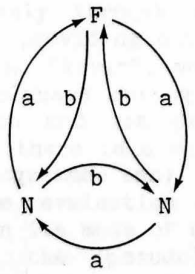

The limit on the number of states is introduced by reducing the definition of next-integer to:

```
next-integer(1,2) ->
next-integer(2,3) ->
```

By starting from solution(x), x becomes:



which is the automaton in (19).



(19)

The strings accepted by this automaton are those whose number of occurrences of a's is the same as the number of occurrences of b's, modulo 3. The reader famliar with Prolog will notice the small number of non-deterministic situations this program will confront.

ACKNOWLEDGEMENTS

Thanks are given: to A. Porto for the numerous comments he made on this paper, to M. Van Caneghem for making available the first Prolog interpreter that allowed me to test examples on infinite trees, and finally to J. Cohen for the help provided in giving the final touches to the manuscript. This work was sponsored by the CNRS grant "ATP Informatique 1978".