1978

CW16

INTELLIGENT BACKTRACKING FOR AN INTERPRETER OF HORN CLAUSE LOGIC PROGRAMS.

Maurice Bruynooghe
Aspirant NFWO
Applied Mathematics and Programming Division
Katholieke Universiteit Leuven
Belgium

ABSTRACT.

Intelligent backtracking for an interpreter of Horn clause logic programs.

An interpreter of Horn clause logic programs is given a goal statement $<-A_1,A_2,\ldots,A_n$ $(n >= 1)$ and a set of Horn clauses of the form $B<-B_1,\ldots,B_m$ $(m >= 0)$ and attempts to derive a sequence of goal statements ending in the empty goal statement.

The basic cycle of the interpreter is :

- select a literal A_i in the current goal statement.
- select a Horn clause $B<-B_1,\ldots,B_m$ such that A_i and B have a most general unifier (θ) and derive a new goal statement : $<-(A_1,\ldots,A_{i-1},B_1,\ldots,B_m,A_{i+1},\ldots,A_n)\theta$

In general, different Horn clauses match with A_i. This non-determinism can be represented as an or-tree. At the implementation level, this non-determinism is usually handled by backtracking : only one path at a time is explored in the or-tree. A simple backtracking system such as in the current PROLOG respects the total ordering provided by the selection function and returns to the previous goal statement when the current one has no solutions. However, possibly the previous goal statement has no solutions for just the same reasons.

In this paper we define a partial ordering over the different steps in a derivation. This partial ordering allows us to find a goal statement that cannot be unsolvable for the same reasons as the current one ("backtrackpoint").

Moreover, this partial ordering allows us to reorder the steps of the derivation, without losing any possible solutions, such that not all steps executed since the "backtrackpoint" must be undone.

This partial ordering also reduces the amount of computation necessary to derive all solutions when the derivation of a solution contains independent subproblems.

1. Horn clause logic programs.

1.1. Syntax and declarative semantics.

A Horn clause program comprises a set of procedure declarations and a "main program" or "goal statement". Procedure declarations are expressions of the form $B<-A_1,...,A_n$ $(n >= 0)$ where $B,A_1,...,A_n$ are atomic formulas. They can be read as logic statements i.e. for all values of the variables, B is true if $A_1\&...\&A_n$ are true.

An atomic formula is an expression of the form R(t1,...,tn) where R is a n-adic relation and t1,...,tn are terms, i.e. functional expressions, constants or variables. We distinguish constants from variables by starting the former ones with an upper case letter and the latter ones by a lower case letter.

A declaration $B<-A_1,...,A_n$ where the atom B has a relation name R is the declaration of a procedure for the relation R.

The main program is given by an expression of the form $<-A_1,...,A_n$ $(n >= 1)$ where the "procedure calls" A_i are again atomic formulas. This main program can be interpreted as a request to find a constructive proof of $\exists\, x1,...,xm\ (A_1\ \&...\&\ A_n)$ where x1,...,xm are the variables in $A_1,...,A_n$.

1.2. The procedural interpretation.

The main program $<-A_1,...,A_n$ is evaluated as follows : we select an arbitrary procedure call $A_i=R(t1,...,tp)$. We invoke a procedure for the relation R of this call, by searching for a declaration $R(t'1,...,t'p)$ $<-B_1,...,B_n$ such that R(t1,...,tp) and R(t'1,...,t'p) unify with a most general unifier θ. We transform the main program into a new set of procedure calls $<-(A_1,...,A_{i-1},B_1,...,B_m,A_{i+1},...,A_n)\theta$ which is the new state of the computation. This process is repeated until we reach an empty set of procedure calls. The composition $\theta_1.\ \theta_2.....\ \theta_k$ of the unifying substitutions $\theta_1,...,\ \theta_k$ of this computation, applied to the variables in the initial set of calls, is the output of the computation.

1.3. Non-determinism.

The above evaluation mechanism is non-deterministic in two respects. First, there is, in the current set of procedure calls, the choice of the call to be executed in the next step. Control over this selection is of crucial importance for efficiency of the computation and - because the really intelligent scheduler is still unknown and is not likely to be found in the near future - has to be done by the user. In current implementations of the language PROLOG, [1,2,3] selection is done in a strictly left to right order. The user has to adapt his logic programs in such a way that this selection rule gives an adequate efficiency of the computation. Work is under way [4] to provide a more flexible control. This will allow the same overall efficiency with a sometimes much simpler logic program. Notice that the selection rule has no effect on the number of derivable solutions.

The second non-determinism results from the possibility of different procedures matching the selected call. Each of them leads to an alternative branch in the computation. We can explore these branches in parallel, or, we can explore the alternatives sequentially, and find all solutions, using backtracking, if for the strategy used to select the call, each branch terminates with a solution or a failure. Observe that the same restriction on the selection of calls is necessary in order to terminate a parallel execution when searching all possible solutions.

Where a naive backtracking system will rigorously explore all

alternatives, a more intelligent one will learn from previous failures and successes how to get a faster exploration of the remaining alternatives. In this paper we study such an intelligent backtracking system.

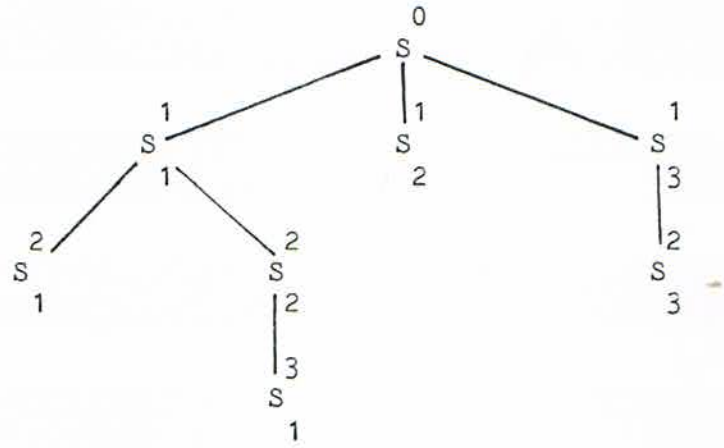
The same subject has been studied by some MSc students at Imperial College in London in a project led by Robert Kowalski, a project that had its origins in the study of Sussman's HACKER. Intelligent backtracking in connection graphs is discussed by Cox and Pietrzykowski [5,6].

2. Naive backtracking.

An interpreter is given a set S of procedure calls of the form $\leftarrow A_1,\ldots,A_n$ (the "goal statement"). Its basic cycle is the following :

- select a literal A_i of S
- select a procedure $B\leftarrow B_1,\ldots,B_m$ whose heading B matches with A_i (B and A_i have a most general unifier θ) and derive the new set $(\leftarrow A_1,\ldots,A_{i-1},B_1,\ldots,B_m,A_{i+1},\ldots,A_n)\theta$.

In general, k procedures match the selected literal A_i and thus, k new states are derivable. We can represent these alternative solutions as branches in an or-tree e.g. :



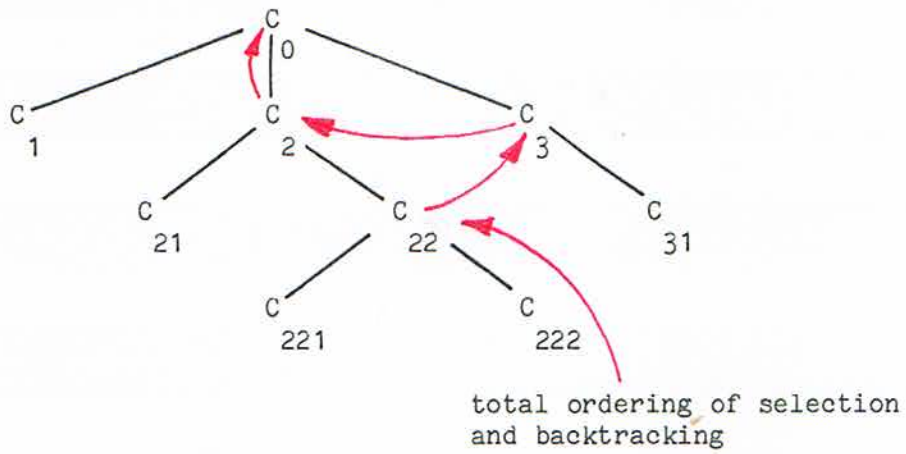
The terminal nodes are either solutions (the empty set of calls) or failure nodes (none of the procedures matches the selected call). Backtracking explores this or-tree using a depth-first, left-to-right search strategy. Once all the solutions to some S^i_j descendant of S^{i-1}_k are found, the system restores S^{i-1}_k and if there are still some untried alternatives to match the literal selected in S^{i-1}_k, one of them is tried and S^i_{j+1} is derived. If there are no more alternatives, it means all solutions to S^{i-1}_k are found and the system backtracks to the previous level.

The shortcoming of this system is that several states occurring in different branches of the or-tree share some procedure calls, and, in executing these calls, the same computations are done in each branch. To be more specific, consider an initial set of calls $\leftarrow A,B$ where the literals A and B do not share any variable. Suppose the selection rule is such that the call A is completely executed before the call B is selected. A naive backtrackingsystem will derive a first solution for A characterized by a substitution θ_1, then will execute the call $B\theta_1$ and will derive all solutions to this call, then will backtrack to A, deriving a second solution, characterized by θ_2 will execute the call $B\theta_2,\ldots$ until all solutions are derived. However, because A and B do not

share any variables $B\theta_1 = B\theta_2 = \ldots = B\theta_n = B$. The system will execute the same call B n times (n being the number of solutions for A); in other words, the same node will occur in n different branches of the or-tree. A more intelligent system should recognize this independence, should execute both calls A and B only once and should combine the n solutions $\theta_1,\ldots,\theta_n$ for A with the m solutions $\sigma_1,\ldots,\sigma_m$ for B to obtain the $n * m$ solutions $\theta_i.\sigma_j$ for the initial problem <-A,B. In the special case where m = 0, such an intelligent system will not attempt to remedy the failure of B by deriving another solution for A, but will conclude the unsolvability of the total problem.

The current state of a computation can be characterized by an and-tree where the calls of the original goal statement are the branches of the root node. At each step, a leaf is selected and becomes the root of the calls in the body of the applied procedure. The order in which the calls are selected is, in a naive system, also the order in which backtracking is performed and characterizes the whole computation.

Example :



We can think of the computation as starting with a fictitious goal statement $<-C_0$. The original goal statement is $<-C_1,C_2,C_3$. The order of selection is C_0,C_2,C_3,C_{22}. Backtracking occurs in the reverse order. The current goal statement is $<-C_1,C_{21},C_{221},C_{222},C_{31}$. Once all solutions to the current goal statement are found, the system restores $<-C_1,C_{21},C_{22},C_{31}$. Augmented with, for each call, the list of untried procedures and the applied substitution, this tree defines the whole computation.

The father of a call is the call by which the former is created i.e. C_2 is the father of C_{21} and C_{22}.

The offspring of a call consists of all calls in the subtree with that call as root i.e. the set $\{C_{21},C_{22},C_{221},C_{222}\}$ is the offspring of C_2.

3. A more accurate backtracking on failure.

A call A on some relation R has, when it is executed, a certain pattern (its arguments have a certain value). This pattern (or parts of it) determines the definitions matched by the call. The actual expression of the pattern is determined by some of the calls already executed, especially by its father and other ancestors but, eventually, also by others. In executing a call, the system searches for the first definition matching the call. Assume that the system can determine which of the previously executed calls generate parts of the pattern which are essential for the eventual mismatches and for the first

match. Let D_A denote such a set for a call A. We say that a call "depends" on all calls in its D-set. We delay the precise definition of this dependency relation until the next section, but we can already make some observations about it :

1. Including too many calls in the D-set of a call cannot cause the loss of solutions; however, we can expect that a smaller D-set gives more accurate backtracking.
2. Any reasonable definition of the dependency relation, has to be transitive. Indeed, when the first definition matching a call A critically depends on the pattern generated during the execution of a call B, and the same relationship exists between the calls B and C, then the call A must also depend on the call C. As a consequence, it is sufficient to keep track of the calls on which each call "depends directly" (those not induced by transitivity).
3. A call is at least dependent on its father (and by transitivity on its other ancestors): indeed, a call only exists because its father has been executed with a particular definition.

Each call in a D-set has a corresponding goal statement, i.e. the goal statement in which that call has been selected.

With all these observations in mind, we can state what the system has to do when it selects some call A which does not match any procedure definition. A naive backtracking system will restore the previous goal statement say G_B and will try another procedure definition for the call B. However, undoing B is insufficient when B does not belong to the set D_A. Indeed, in this case, the system did not need the substitutions generated by the call B to detect the unsolvability of the call A and, thus, changing these substitutions cannot make the call A solvable. We conclude that we have to backtrack further and to restore the goal statement G_C with C the most recently executed call in D_A. Indeed, restoring any of the goal statements between G_C and G_A cannot make the call A solvable, and we do not have to explore the corresponding branches of the or-tree.

Notes :

1. The most recently executed call in D_A is one of those on which A depends directly.
2. Either G_C still contains the call A, eventually with a less specified pattern, or C is the father of A (because the father of A belongs to D_A).
3. The goal statements corresponding to the set D_A are the only points in the or-tree where possibly successful alternatives can start. By alternative, we mean a point which can lead to a different pattern for the call A or no call A at all, in other words a goal statement which <u>can</u> lead to derivations where the same conflict does not occur. On the contrary <u>all</u> goal statements in the or-tree corresponding to a <u>solution</u> are branching points which can lead to different solutions.

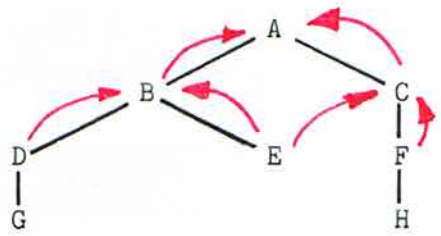
Having rejected the current solution for C, the system searches another definition matching C and, eventually, tries the call A again. To assure that the system - when it, using the other definitions for C, still fails to derive a solution - also tries a new start from the goal statements corresponding to the remainder of the set D_A we have to update the D-set of the call C. We have to replace D_C by the union of D_C and $\{ D_A - C\}$. This extended set is still in agreement with the intended definition of the D-sets, indeed it contains all calls which contribute to the generation of patterns essential in the rejection of all tried definitions for the call C. Now, one of these definitions is rejected because it causes a conflict with the call A. Also the pattern of A is essential in this conflict and thus, those calls creating the critical parts of

that pattern (those causing the failure of all available definitions) must be in the D-set of the call C.

This update of the D-set brings us back to the initial assumption about the D-set (now for the call C instead of A) and thus, the above reasoning can be repeated.

Systematic updating of the D-sets involved - when backtracking and when trying new definitions for a call - assures that the system, in search for a first solution, will try all valuable branching points of the or-tree and will prune all others. As the system has learned from previous failures, these other branches are doomed to fail.

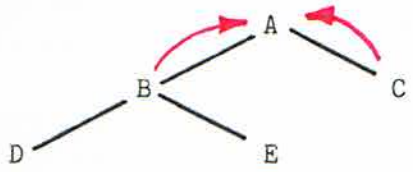
Assuming that the calls in the original goal statement depend on some fictitious node, the direct dependency defines a partial ordering over the nodes of the and-tree. We sketch a simple example (in addition of the and-tree, we indicate the direct dependencies by arrows).



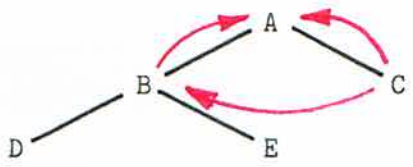
Sequence of goal statements :

G_B=<-B,C
G_C=<-D,E,C
G_D=<-D,E,F
G_F=<-G,E,F
G_E=<-G,E,H

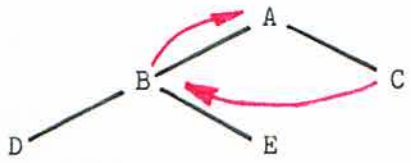
When the call E in G_E fails, the system restores G_C (C being the most recent call in D_E = {B,C})



updates D_C : $D_{Cnew} = D_{Cold} \cup \{ D_E - C\}$



removes direct dependencies induced by transitivity and tries another definition for the call C.



4. Defining dependency.

4.1. A special case : naive backtracking.

The assumption that every call depends on all previously executed calls leads to the total ordering of a naive backtracking system. Indeed, as a consequence of transitivity, each call depends directly on only one call, the one previously executed, and thus backtracking always restores the previous goal statement.

4.2. A definition based on the unification algorithm.

It is the unification algorithm which decides whether or not a call matches a certain definition. In doing so, the unification algorithm does not need access to all parts of the pattern. As soon as it finds a mismatch, it does not need the remainder of the pattern. Also in unifying a term with a variable, it does not need the components of the term (as far as the occurcheck is unnecessary). The unification algorithm has access to all components generated by calls on which the current call depends. Initially, the current call depends on its father (because as we already said, the call only exists as a consequence of executing the father with a particular procedure definition) and, by transitivity, on those calls on which the father depends and the unification algorithm has access to all substitutions generated by these calls. As soon as the unification algorithm needs access to other substitutions, dependencies are added.

An example :

A call : P (x)

```
        | (1)
        f ( k , l )
            | (2) | (3)
            A     f ( m , n )
                      | (4) | (5)
                      B     f ( p , q )
```

The substitutions x <- f(k,l) has been generated by the call (1), k <- A by (2), l <- f(m,n) by (3), m <- B by (4) and n <- f(p,q) by (5). Suppose the call P initially depends on (1) but not on (2) (3) (4) and (5). As a consequence, while executing the call P the unification algorithm initially only has access to the substitution x <- f(k,l). It knows k and l are bound but it does not know to which terms.

Assuming that the attempted definition is :

P(f(r,f(s,t))) <- Q(r,s)), P(f(s,t))

the system has to unify the bound variable x with the term f(r,f(s,t)). To do so, it has to know the value of x. Because the call depends on (1) the unification algorithm can access f(k, l) without adding new dependencies. The functors are the same and the system has to unify their arguments. It

has to unify the bound variable k with the free variable r. Because this is the first occurrence of r, unification must be possible (whatever the value of k), and the substitution r <- k can be generated without accessing the value of k, thus without making the current call dependent on (2). In the next step the bound variable l has to be unified with the term f(s,t). To do so we have to access the first approximation of l : f(m,n).

The necessity to access this structure makes the current call P dependent on (3). In the next step we have to unify the free variable s with the bound variable m and the free variable t with the bound variable n. Again, because these are the first occurrences of s and t, they cannot occur inside the terms to which m and n are bound, and the substitutions s<-m, t<-n can be generated without accessing the values of m and n, thus without making the current call dependent on (4) and (5).

Giving the substitutions generated by the call P a label (6), the children of P become :

```
Q ( r , s )   and     P ( f ( s , t ))
  | (6) | (6)               | (6) | (6)
  k     m                   m     n
  | (1) | (4)               | (4) | (5)
  A     B                   B     f ( p , q )
```

They depend initially on their father (6) (the call P) and the calls on which the father depends i.e. (1) and (3).

Notes :

1. An implementor should consider possible optimisations for example, because the children depend on (6), they can as well be replaced by

```
Q ( k , m )   and     P ( f ( m , n ))
  | (2) | (4)               | (4) | (5)
  A     B                   B     f ( p , q )
```

2. In unifying a free variable with a bound variable, when the occurcheck is necessary, the current call becomes dependent on all calls generating parts of the term to which that bound variable is bound. As a consequence, to get accurate backtracking, it is necessary to perform the occurcheck only where it really is necessary. Most implementors of languages based on logic do not apply an occurcheck at all, however, some programs have been written where the occurcheck is essential [7]. Further in this paper, we argue that dependencies caused by occurchecks have only to be added when the occurcheck causes the failure of the unification.

In detecting a mismatch, the unification algorithm can have some choice because there can be more than one disagreement between the call and the definition. In such cases it seems preferable to access structures generated by the least recent calls, they will give the deepest backtracking and the highest number of pruned branches in the or-tree.

We illustrate the backtracking behaviour with the n-queens problem for n=4. The initial goal statement is :

```
<- Perm (4.3.2.1.Nil, 1) Pair (4.3.2.1.Nil, 1, q) Safe (q)
```

The execution order is strictly left to right. The call Perm generates a permutation 1 of the list 4.3.2.1. Nil, e.g. 1.2.3.4. Nil, the procedure Pair combines its first two argument lists into a queenboard q with one queen in each row and one queen in each column e.g. p(4,1).p(3,2).p(2,3).p(1,4).Nil.

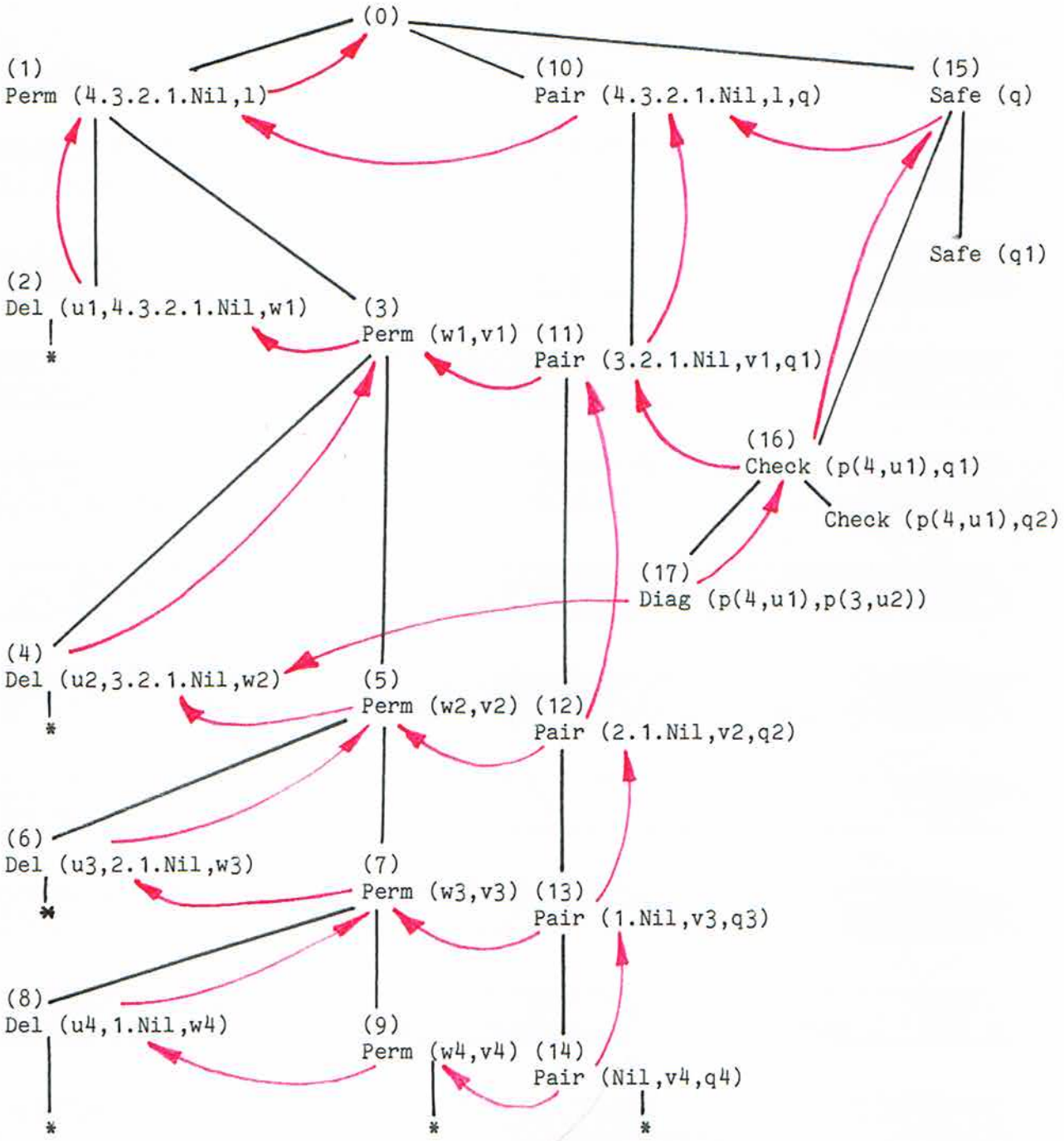
The procedure Safe rejects the queenboard and causes backtracking when it finds two queens on the same diagonal. A naive backtracking system will systematically generate all permutations until some permutation passes all checks created by Safe. In our system, to detect a conflict between for example the first and the second queen, Safe needs no access to the remainder of the permutation, and thus is not dependent on the non-deterministic procedures generating that remainder. As a consequence it will not try to solve the conflict by generating other remainders of the permutation but will backtrack immediately to the procedure generating the second queen. The interested reader finds a detailed analysis of this behaviour in the remainder of this section.

The complete program is :

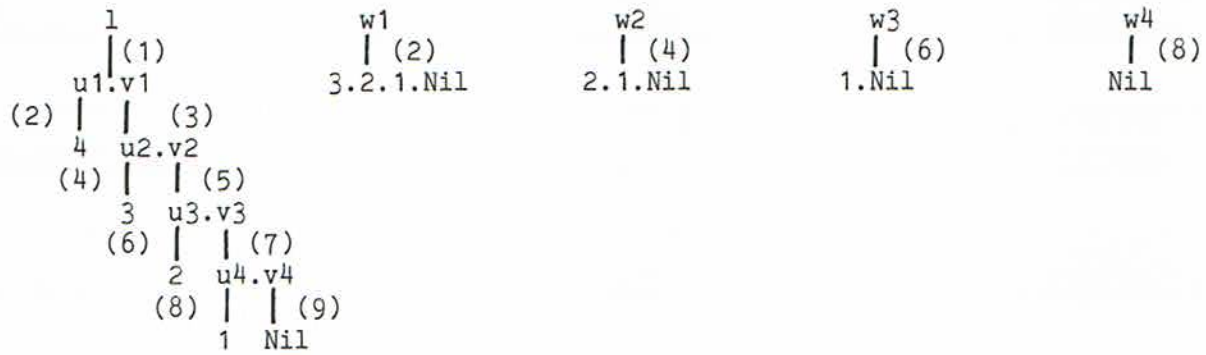
```
Perm (Nil, Nil) <-
Perm (x.y, u.v) <- Del (u, x.y, w) Perm (w, v)
Del (x, x.y, y) <-
Del (u, x.y, x.v) <- Del (u, y, v)
Pair (Nil, Nil, Nil) <-
Pair (x.y, u.v, p(x,u).w) <- Pair (y, v, w)
Safe (Nil) <-
Safe (p.q) <- Check (p, q) Safe (q)
Check (p, Nil) <-
Check (p, q.r) <- Diag (p, q) Check (p, r)
```

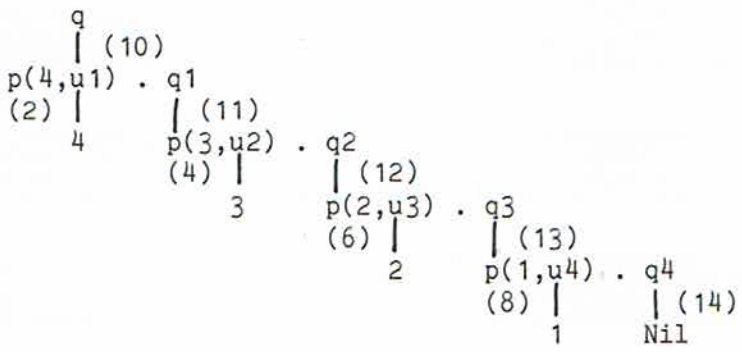
For simplicity, we assume Diag defined as a set of assertions about queens not on the same diagonal.

```
Diag (p(1,1), p(1,2)) <-
Diag (p(1,1), p(2,3)) <-
        .
        .
        .
```

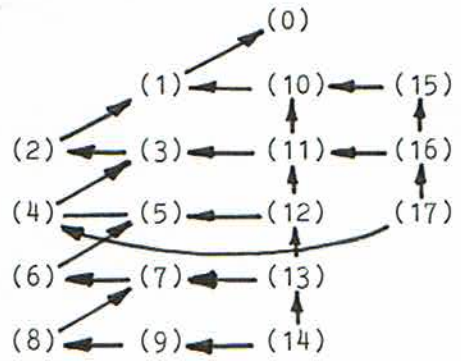


And-tree and dependecy-graph at the point where call (17) causes a failure.

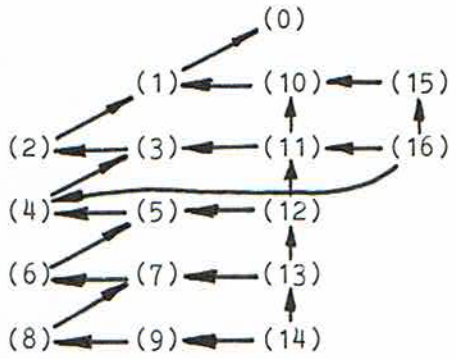




Substitutions generated by the different calls.



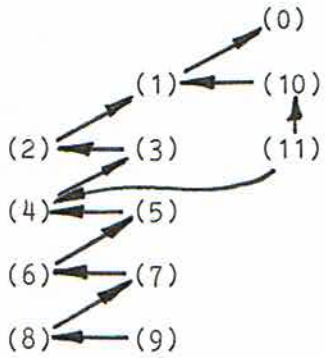
Direct dependency-graph .
(17) fails because of the conflict between the first and the second queen.



The goal statement prior to the execution of call (16) is restored, and the D-set of (16) is updated.
No other definition for (16) is available and the system backtracks further.

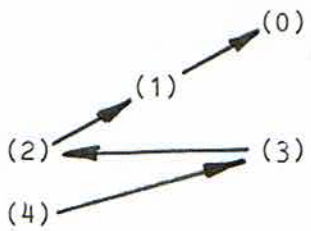
- restore the goal statement prior to the execution of (15); update D-set
- restore the goal statement prior to the execution of (11); update D-set

There is still no alternative definition available and the dependency-graph becomes as follows :



- restore the goal statement prior to the execution of (10); update D-set
- restore the goal statement prior to the execution of (4); update D-set

The dependency-graph is now :



(4) (Del(u2,3.2.1.Nil,w2)) has an alternative definition. The system has backtracked correctly to the point where the second queen was chosen. A naive backtracksystem would try to find other solutions for the last queen,

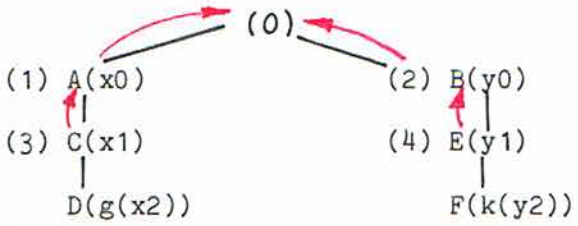
the last but one, ... !

5. On the order of execution.

We claim that any order of execution allowed by the partial ordering leads to the same current goal statement and the same dependency-graph.

We give a simple example :

```
A(f(x)) <- C(x)
C(x) <- D(g(x))
B(h(y)) <- E(y)
E(y) <- F(k(y))
<- A(x0) B(y0)
```



The sequence of calls (1) (2) (3) (4) leads to the goal statement <-D(g(x2)),F(k(y2)).

We claim that any sequence allowed by the partial ordering of the dependency-graph, i.e. (1) (3) (2) (4) and (2) (4) (1) (3), leads to the same goal statement and the same dependency-graph.

We know from resolution theory that the order of different resolution steps does not change the final clause.

We only have to show that the dependencies are also unchanged. We do this by arguing that two successive steps, not dependent on each other, can be interchanged without causing changes in the dependency-graph.

Suppose we execute a call A. It matches the i-th definition and results in a substitution $\sigma = \{x_k <- s_k\}$. The next call executed is a call B. It matches the j-th definition and results in a substitution $\theta = \{y_k <- t_k\}$.

Suppose B is not dependent on A.

Now we execute B before A. Because B was not dependent on A, it did not look at the substitution σ generated by A and consequently it does not matter for B whether or not the substitution σ is present. B will again match the j-th definition with the same substitution θ . Executing A after B, the unification algorithm surely can, as before, reject the first i - 1 definitions without looking at θ and this will result in the same dependencies as before. Because interchanging the resolutionsteps does not change the final clause, the call A must match the i-th definition, but, will the unification algorithm look at the substitution θ ? Suppose it does. Then the unification algorithm must find some pair of the form x - s to be unified, with x a variable bound by the

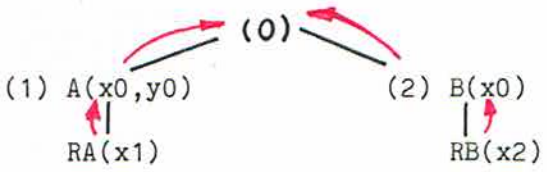
substitution θ(x<-t $\in \theta$) and s bound to some term. Indeed, only in such a situation must the unification algorithm look at θ and does A become dependent on B. This is not possible, indeed, in that case executing A before B must have resulted in x<-s $\in \nabla$ and, during the execution of B, a pair x - t to be unified. At that point, either t<-x was generated or the unification algorithm had to look up x and B becames dependent on A. This contradicts the assumptions that B does not depend on A and that x<-t$\in\theta$. Thus while executing A after B, the unification algorithm does not need θ and will generate the <u>same</u> substitution and the <u>same</u> dependencies as before. Because ∇ and θ are the same as before, all calls executed <u>after</u> A and B get the same dependencies and the interchanging of A and B does not affect the dependency-graph.

There is a weak point in the above reasoning : we did not mention occurchecks. Maybe, changing the execution order causes different occurchecks and these occurchecks in turn cause different dependencies.

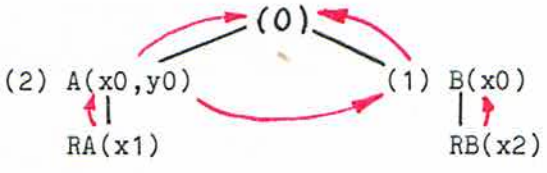
An example :

```
A(x1,x1) <- RA(x1)
B(f(x2)) <- RB(x2)
<- A(x0,y0) B(x0)
```

Starting with the call A we get x1<-x0 and we have to unify y0 with x1. Before generating the substitution y0<-x1, we have to verify that y0 does not occur inside the substitution for x1. Then we execute the call B and we get the substitution x0<-f(x2).
The dependency-graph is :



Starting with the call B we get the substitution x0<-f(x2) then while executing A, we have to unify x0 with x1. Because it is the first occurrence of x1, we can generate x1<-x0. Then we have to unify y0 with x1 : we have to verify that y0 does not occur in the substitution for x1. For this test, we need to access also the substitution x0<-f(x2) thus the call A becomes dependent on B : we get a different dependency-graph.



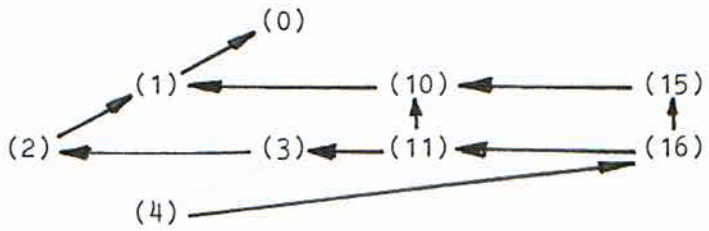
We can solve this problem by some subtle reasoning. As we already mentioned in the previous section, we should only execute the occurchecks which are really necessary. In fact the only necessary occurchecks are those causing the failure of the unification. Now, when changing the order of execution, we know that the derivation is possible and therefore the occurchecks cannot cause a failure and are unnecessary. Thus the above reasoning was correct and

changing the order of execution - respecting the partial ordering - does not change the dependency-graph. We even can apply this observation about the necessity of the occurcheck with retrospective effect, i.e. when we detect that it does not cause a failure, we conclude we should not have done it and we leave the dependency-graph unchanged. We only add the dependencies caused by an occurcheck when it causes the failure of the unification.

As a consequence of the property that we can change the order of execution, when we backtrack, we can choose between any of the calls on which the failing call depends directly. We undo the selected one, say A, and of the calls executed after A, we have only to undo those dependent on A. Then we make A directly dependent on the other calls on which the failing call depends directly, remove the direct dependencies induced by transitivity of others and try another definition for the call A.

When there is a good selection function available, it seems reasonable to backtrack to the most recent call in the direct dependency relation. Is the selection function rather poor, then it can be useful to analyse the effect on the size of the search space of the different possible "backtrackpoints". Eventually, the selection strategy could learn from this analysis.

In the queensexample at the end of the previous section, the failing call (17) depends directly on (4) and (16). Backtracking to (4) yields the graph :



This backtracking is preferable to the one used, because both end with the application of another procedure for call (4), but here, a greather part of the computation has been saved.

6. Finding all solutions.

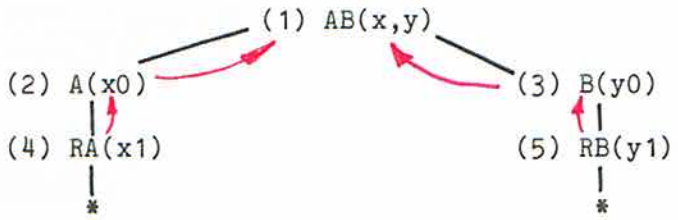
As we already briefly mentioned in the introductory sections, to find all solutions when there are different independent subproblems, each having a set of solutions, a backtracking-system will execute some subproblems several times. The goal of this section is to avoid this inefficiency. We introduce our method by a simple example.

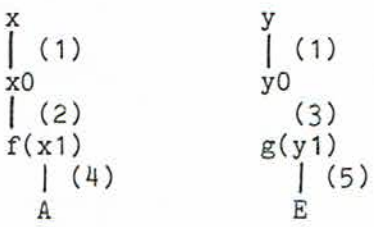
The program :

```
<- AB(x,y)
AB(x,y) <- A(x) B(y)
A(f(x)) <- RA(x)
A(g(x)) <- QA(x)
RA(A) <-
RA(B) <-
QA(C) <-
B(g(y)) <- RB(y)
B(k(y)) <- QB(y)
RB(E) <-
RB(F) <-
```

QB(G) <-

Assume a first solution has been found :



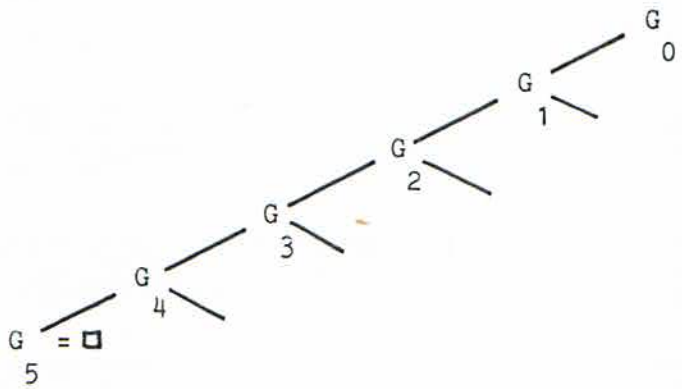


This first solution {x<-f(A), y<-g(E)} can be derived by all sequences allowed by the partial ordering. Let us choose one of these orderings i.e. (1) (2) (3) (4) (5).

The sequence of goal statements is then :

G_0 = AB(x,y)
G_1 = A(x0) B(y0)
G_2 = RA(x1) B(y0)
G_3 = RA(x1) RB(y1)
G_4 = RB(y1)
G_5 = □

To find all solutions, we have to explore all branches of the corresponding or-tree :



We can explore this or-tree bottom up i.e. find all solutions for G_4 then

all solutions for G_3 ,... then all solutions for G_0. To improve the efficiency, we will divide, where possible, these goal statements into independent subproblems. We can solve these subproblems independently of each other and obtain the solutions for the complete goal statement by making the cross-product of their solutions. In fact, we will not explicitly compute the set of solutions for each successive goal statement. As the example will illustrate, we only compute a solutionset for well chosen subproblems (See the "merge procedure" below.)

G_4 (=RB(y1)) consists of a single call, all its solutions can be found simply by bracktracking over the call RB(y1).

By expanding G_4 into G_3 , another call RA(x1) enters the goal statement. The dependency-graph shows us that both are independent. All solutions for G_3 could be found by :

- searching for all solutions of each subproblem, using normal backtracking; this yields [{y1<-E}, {y1<-F}] for RB(y1) and [{x1<-a}, {x1<-B}] for RA(x1).
- making the cross-product of both sets; this yields : [{y1<-E, x1<-A}, {y1<-F, x1<-A}, {y1<-E, x1<-B}, {y1<-F, x1<-B}].

In the next step, the subproblem RB(y1) expands into the subproblem B(y0) (the call being replaced by its father) while the other subproblem remains unchanged. Still, the solutions for each of the subproblems can be obtained by backtracking. Similarly, while moving to G_1 , the call RA(x1) expands into A(x0) and we have two independent subproblems A(x0) and B(y0) : the solutions for each of them can be found by backtracking. The last step is more interesting : here, both independent subproblems merge into the single call AB(x,y). Systematic backtracking is inefficient for this goal statement. We apply the following "merge procedure" :

1. Find all solutions for the subproblem A(x0) i.e. [{x0<-f(A)}, {x0<-f(B)}, {x0<-g(C)}]. In general, this is done by recursively applying the "find-all-solutions" methods, which in this example simplifies to systematic backtracking over the nodes (2) and (4). In fact, the system alternates between :
   - Find a new solution : systematic backtracking over all nodes in the current solutions but intelligent backtracking over the new nodes (those entering the graph when it grows again : for these nodes we again search a first solution).
   - Find all solutions for the subproblem.

2. Similarly, find all solutions for B(y0) i.e. [{y0<-g(E)}, {y0<-g(F)}, {y0<-k(G)}].

3. Compute all solutions for A(x0), B(y0) by making the cross-product of both sets : i.e. [{x0<-f(A), y0<-g(E)}, {x0<-f(A), y0<-g(F)}, ...].

4. The "current solution set" for AB(x,y) is obtained by making the composition of the substitution applied on the call ({x<-x0, y<-y0}) with the substitutions of the above set : i.e. [{x<-f(A), y<-g(E)}, ...] (we are only interested in substitutions for the variables in the goal statement AB(x,y)).

5. We can reduce the dependency-graph and summarize the obtained results in a "reduced node". The reduced node 1_* replaces all calls involved ((5) (4) (3) (2) (1)) and has, instead of a single substitution, a set of substitutions (the "current solution set") characterizing all obtained results.

The graph becomes :

(1_*)
AB (x,y)

A single reduced node with a current solution set and a list of untried procedure definitions (in this example empty).

Now, we determine where the next "merge" operation must be performed, or, as in this example, start to collect the solutions for the initial goal statement :

- take the current solutions set
- find all remaining solutions, by alternating between :
  - find a new solution
  - and - find all solutions

Where a backtracking system directly starts collecting all solutions for the initial goal statement, the above procedure collects solutions for well chosen subproblems, and merges these subproblems into bigger ones until finally the original problem with its solutions is obtained.

The merge of subproblems is not always as simple as in the example above. There, in expanding G_1 into G_0, the entering call AB replaced its two independent children A and B. In general G_j consists of an independent subproblem $A_1,\ldots,A_n$ $(n >= 1)$ and another independent subproblem $B_1,\ldots,B_m$ $(m >= 1)$. By extending the goal statement G_j into G_i, a call C enters. C is such that some of the A_i as well as some of the B_j depend on it. The new goal statement consists of the calls $\{A_i\} \cup \{B_j\} \cup \{C\}$ - { children of C }. C becomes the reduced node, its current solution set is obtained as described in the above example. Unlike the simple case, those A_i and B_j which are <u>not</u> children of C remain in the graph and depend on C : indeed, they have to be executed again when the backtracking mechanism tries the remaining procedure definitions on C.

When a first solution is found and we start to collect all solutions, we do not have to decide immediately which or-tree we will use in the derivation of all solutions. We can take this decision step by step, we only have to be careful to respect the partial ordering. Respecting the partial ordering means that the call entering the goal statement has no other call depending on it which is not yet in the goal statement. Manipulation is simpler and the alternatives are clearer when we perform an update of the dependency-graph which is similar with the one used in section 3. Initially, each subproblem consists of a single call with a certain D-set. This call is named the <u>representative</u> call of the subproblem. When a subproblem is extended, the entering call becomes the new representative call and its D-set is updated with the D-set of the old representative call; similarly, in the case of a merge, the entering call becomes the representative call and its D-set is updated with the D-sets of the representative calls of the merging subproblems (the representative call becomes the reduced node). Now, the dependencies between the other calls in the subproblems and calls not yet in the goal statement can be ignored and, in each step, to extend the goal statement, we can choose between the calls which are such that the only calls dependent on them are representative calls.

We now give a non-trivial example.

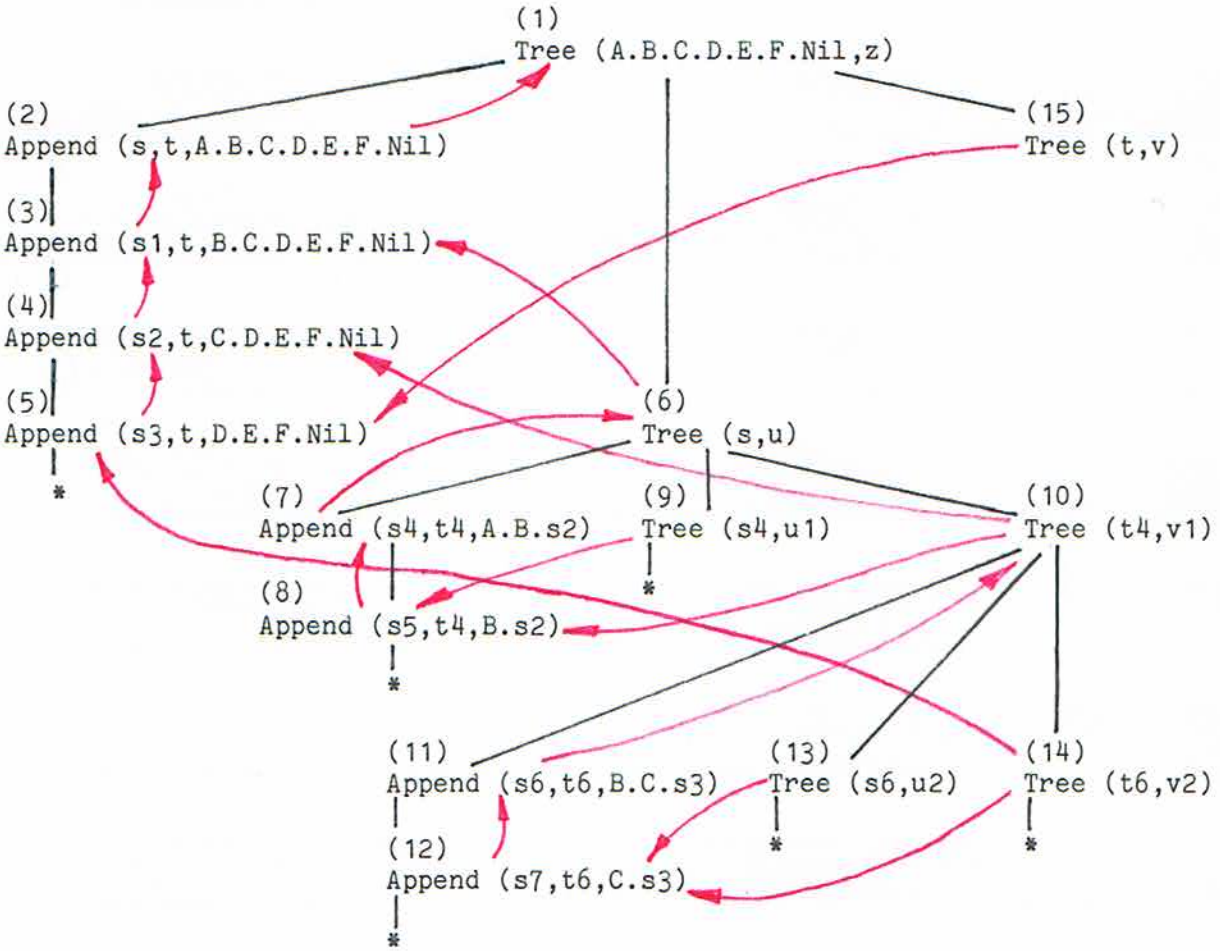
Example :

A program to compute all binary trees having a certain leaf profile.

- x.y represents a list with first element x and remainder y.
- Nil represents an empty list.
- x*y represents a binary tree with x and y subtrees.

```
Tree (x.Nil, x) <-
Tree (x.y.z, u*v)<-Append (s, t, x.y.z) Tree (s, u) Tree (t, v)
Append (Nil, x, x) <-
Append (x.u, v, x.w)<-Append (u, v, w)
<- Tree (A.B.C.D.E.F.Nil, z)
```

We give the computation of a first solution in a strictly left to right execution order. As first definition for "Append" we use the one that is best suited for our purposes : dividing the six-element list into two tree-element lists. We only give the complete computation of the left subtree, the right subtree is similar.



And-tree and dependency-graph.

Substitutions made during the execution of the different calls :

```
1 : {z<-u*v}
2 : {s<-A.s1}
3 : {s1<-B.s2}
4 : {s2<-C.s3}
```

5 : {s3<-Nil, t<-D.E.F.Nil}
6 : {u<-u1*v1}
7 : {s4<-A.s5}
8 : {s5<-Nil, t4<-B.s2}
9 : {u1<-A}
10 : {v1<-u2*v2}
11 : {s6<-B.s7}
12 : {s7<-Nil, t6<-C.s3}
13 : {u2<-B}
14 : {v2<-C}

We start with the empty goal statement. It can be extended with one of those calls on which nothing depends e.g. (9) (13) or (14). We choose (14). (14) depends on (12) and (5). However, both have other dependent calls not yet in the goal statement. We concentrate on (12). First we extend the goal statement with (13) (now we have two independent subgoals), then we extend it with (12) : we have to perform a merge.

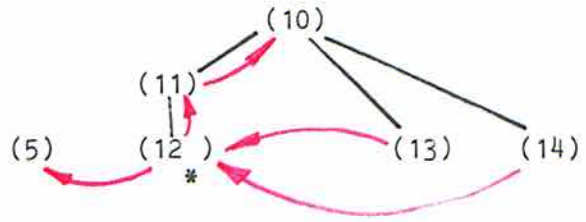
Solutions of (14) : the current solution is {v2<-C}; backtracking yields no other ones.

Solutions of (13) : [{u2<-B}]

Cross-product : [{v2<-C, u2<-B}]

Current solution set of (12) (13) (14) : [{s7<-Nil, t6<-C.s3, u2<-B, v2<-C}]

Part of the reduced graph :



Remark that the calls (13) and (14) stay in the graph and that the "representative" call (12) now depends on (5).

We want to extend the goal statement further with in order the nodes (11), (10), (8). However, before extending it with (8), we first have to extend it with (9), which is independent, and then, we have again to perform a merge.

Solutions for (9) : [{u1<-A}]

Solutions for (10) : current solution [{v1<-B*C}]

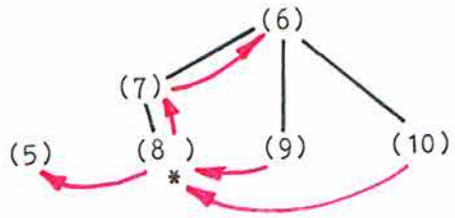
We search other solutions by a systematic backtracking over the nodes (12_*) (11) and (10), while a solution is only found when it also solves the calls (13) and (14) dependent on the reduced node (12_*)

In fact, we search a first solution for the subproblem, but starting with one of the untried procedures for call (12). When we find one, we again start to collect all solutions, (still for the subproblem (10)). This subproblem has no other solutions.

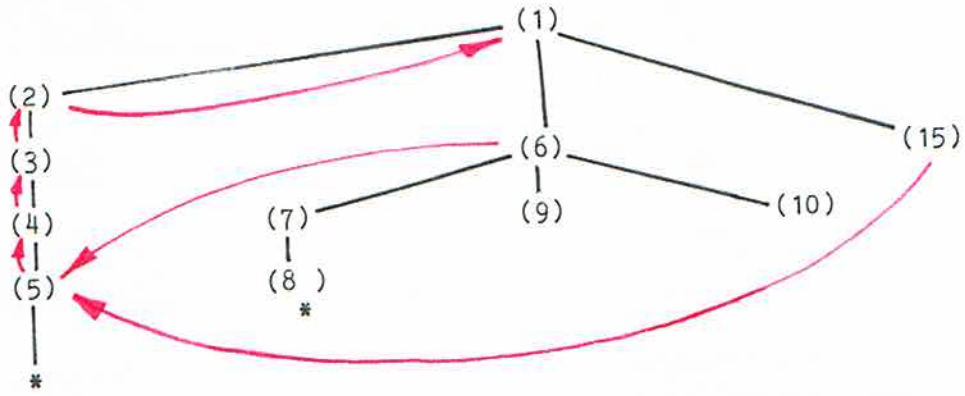
Cross-product : [{ u1<-A, v1<-B*C }]

Current solution set for (8) (9) (10) :
[{s5<-Nil, t4<-B.s2, u1<-A, v1<-B*C}]

Part of the reduced graph :



We can further expand the goal statement with (7) and (6). The graph becomes :



Before extending the goal statement with (5) we have to extend it with the whole subtree of (15), then we have again to perform a merge.

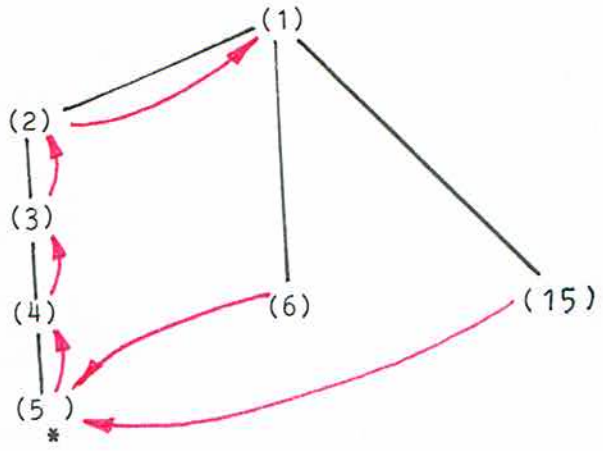
Solutions of (6) : current solution set : [{u<-A * (B*C)}]. Other solutions have to be derived by backtracking over the nodes (8_*) , (7) and (6). This results in a second solution {u<-(A*B) * C}.

Solutions for (15) : similar as for the left subtree, we can derive the solution set [{v<-D * (E*F)}, {v<-(D*E) * F}].

After making the cross-product of both sets, we obtain as current solution set for (5) (6) (15) :

```
[{s3<-Nil, t<-D.E.F.Nil, u<-A * (B*C), v<-D * (E*F)},
 {s3<-Nil, t<-D.E.F.Nil, u<-A * (B*C), v<-(D*E) * F},
 {s3<-Nil, t<-D.E.F.Nil, u<-(A*B) * C, v<-D * (E*F)},
 {s3<-Nil, t<-D.E.F.Nil, u<-(A*B) * C, v<-(D*E) * F}]
```

The reduced graph becomes :



Now the goal statement can be expanded by, in order, 4, 3, 2 and 1 to reach the

initial goal statement.

Its current solution set is :

```
[{(A * (B*C)) * (D * (E*F))},
 {(A * (B*C)) * ((D*E) * F)},
 {((A*B) * C) * (D * (E*F))},
 {((A*B) * C) * ((D*E) * F)}]
```

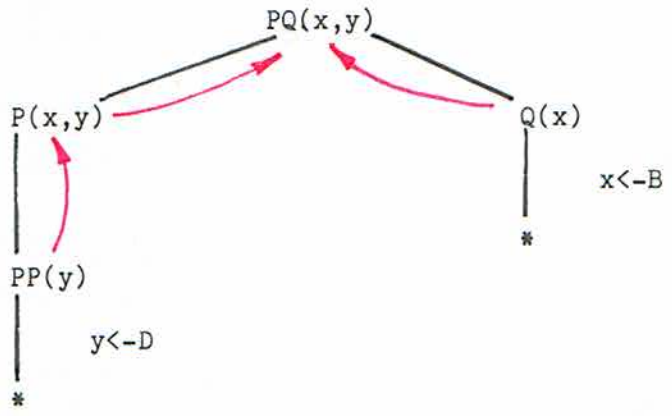
Other solutions will be obtained by systematic backtracking over the nodes (5_*) (4) (3) (2) and (1). The process will start with applying the recursive append definition on call (5) and with deriving (using intelligent backtracking) a new solution. Then, applying successive merge operations, this new solution is transformed into a new solution set and the search for another solution in the new reduced graph starts. This process repeats until all solutions are found.

Some subtle problems can appear from time to time. The dependency-graph is based on the procedure definitions already used, but the ones that have not yet been tried can cause some new dependencies. This gives trouble when a call becomes dependent on a reduced node

We give a simple example :

```
PQ(x,y) <- P(x,y) Q(x)
P(x,y) <- PP(y)
P(A,B) <-
P(B,C) <-
Q(B) <-
Q(C) <-
PP(D) <-
PP(E) <-
```

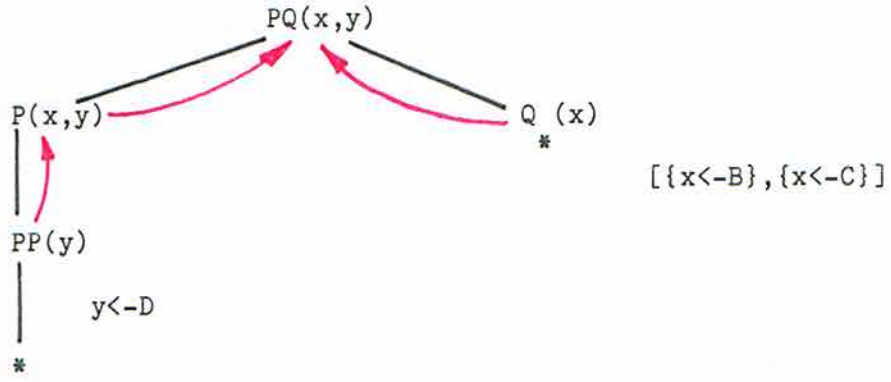
First solution :



As far as this first solution is concerned, P(x,y) and Q(x) are two independent subproblems which merge into the initial goal statement. To find all solutions, we perform a merge

Solutions of Q(x) : [{x<-B}, {x<-C}]

Here we can define an intermediate dependency-graph with node Q reduced and having an empty list of untried procedure definitions.



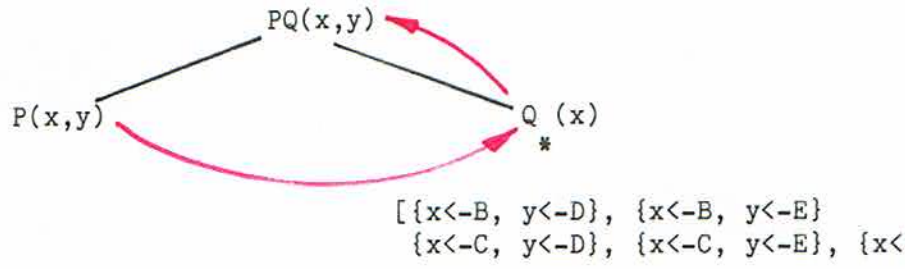
Solutions of P(x,y) :

Backtracking over PP results in the solutions {y<-D} and {y<-E}. Trying the other procedure definitions for P makes P dependent on the reduced node Q.

To find all solutions for P(x,y), Q(x), we have to be very careful. A first set of solutions is found by making the cross-product of the solutions already derived for P and Q :

[{x<-B, y<-D}, {x<-B, y<-E} {x<-C, y<-D}, {x<-C, y<-E}].

Others are found by applying the yet untried procedures for P, to each solution of Q. For x<-B, this results in a solution {x<-B, y<-C}; x<-C results in a failure. This results in the following reduced graph :



Q_* does not change its (empty) set of untried procedures. Instead of deriving the current solution set for PQ, the additional dependency has forced us to first derive a current solution set for Q(x), P(x,y). The further reduction to derive all solutions for the initial goal statement can be done usual.

7. Exploiting determinism.

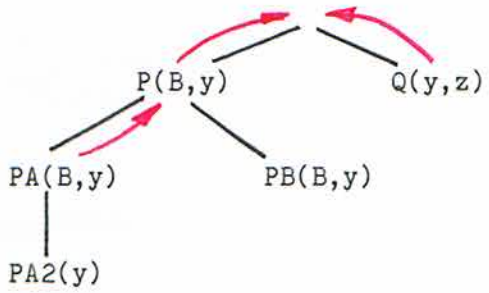
The dependency-graph representing the state of the computation can be simplified by exploiting the present determinism. When we execute a call which depends directly on its father only and there is only one definition between the available definitions matching the call, then we can interpret the execution of this call and the father as a single step in the computation, i.e. obtained by applying a preprocessed procedure on the father. Thus we can remove the node representing the son from the dependency-graph while the list of untried procedures for the father remains unchanged.

A simple example :

```
<- P(B,y) Q(y,z)
P(x,y) <- PA(x,y) PB(x,y)
P(x,y) <- PP(x,y)
PA(A,y) <- PA1(y)
```

PA(B,y) <- PA2(y)

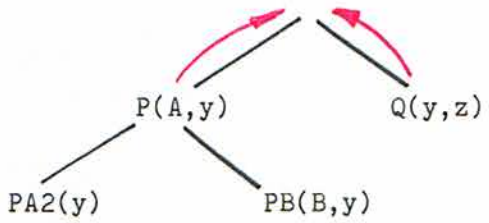
Dependency-graph after executing the calls P and PA :



Preprocessing the call PA in the first definition of P leads to two procedures :

P(A,y) <- PA1(y) PB(A,y)
P(B,y) <- PA2(y) PB(B,y)

In general, preprocessing is not interesting : it leads to a different list of untried procedures (for the call P). However, when only one of the preprocessed procedures matches (as in the above example), then the list of untried procedures remains unchanged and we can derive in a single step the following dependency-graph.



It differs from the first graph only by the fact that the determinate call PA is disappeared. Instead of performing preprocessing, we prefer to perform postprocessing, i.e. when we execute the call PA, we observe that it only matches <u>one</u> procedure, and only depends on its father; we conclude that we could have applied a preprocessed procedure on the father call without changing its list of untried procedures. We apply the preprocessed procedure afterwards : we remove the determinate call from the graph and we extend the substitution of the father call with the substitutions made during the current step.

In fact, it is not necessary for the call to depend directly on its father only. However, when the son also depends on a third call, then we have to make the father dependent on that third call. This probably will result in a less accurate backtracking : it becomes impossible to backtrack from the son to that third call, also a failure of the father caused by for example another son will unnecessarily lead to backtracking towards that third call.

8. Conclusions.

We have defined a partial ordering over the set of calls involved in the execution of a goal statement.

This partial ordering, built by the unification algorithm, allows the accurate determination of all calls which can break a given failure. Moreover,

it allows the reordering of the executed calls such that only a minimal number of calls must be undone during backtracking.

This partial ordering also allows the isolation of independent subgoals. A method to find all solutions, which uses a limited form of parallelism to exploit the detected independence of subgoals is given.

Our method still contains non-deterministic steps :

- When detecting that a call does not match a certain definition, the unification algorithm can sometimes choose which substitutions it will access and thus which dependencies it will add.
- When an unsuccessful call triggers backtracking, the system has to choose one of the calls on which the failing call depends directly.
- While searching all solutions, there can be different calls by which the goal statement can be extended.

All these choices affect the size of the search space and thus the efficiency of the computation. Determining (simple) criteria to make the right choices remains an area of further research. Another interesting question is whether we can develop selection strategies which can learn from the behaviour of the backtrackingsystem. The development of such a learning system could be a major step towards the development of systems which can execute logic programs efficiently without need for any user-specified control information.

9. Acknowledgements.

I am indebted to Robert Kowalski and Yves Willems for their comments on an earlier draft of the paper. This research is supported by the Belgian "Nationaal Fonds voor Wetenschappelijk Onderzoek".

References.

[1] Battani G. and Meloni H.
Interpreteur du langage de programmation PROLOG.
Groupe d'Intelligence Artificielle, Marseille-Luminy 1973.

[2] Warren D., Pereira M. and Pereira F.
PROLOG - The language and its implementation compared with LISP.
Proceedings ACM Conf on "AI and Programming Languages".
Rochester, New York, Aug. 1977.

[3] Bruynooghe M.
An interpreter for predicate logic programs.
Report CW 10, Applied Maths & Programming Division.
Katholieke Univ. Leuven, Belgium Oct. 1976.

[4] Clark K. and Bruynooghe M.
A control regime for Horn clause logic programs.
Forthcoming report.

[5] Cox P. and Pietrzykowski T.
A Graphical deduction system.
Dept. of Computer Science, University of Waterloo, Ontario, Canada, 1976.

[6] Cox P.
Deduction plans : a graphical proof procedure for
the first order predicate calculus.
Dept. of Computer Science, Univ. of Waterloo, Ontario, Canada, 1978.

[7] Tarnlund S.
A logical basis for data bases.
Report TRITA - IBADB 1029, Dept. of Computer Science, Royal Institute
of Technology, Stockholm, Sweden, 1976.