On infinite loops in logic programming

Philippe Besnard

INRIA

UNITÉ DE RECHERCHE
INRIA-RENNES

Institut National
de Recherche
en Informatique
et en Automatique

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

Rapports de Recherche

N° 1096

Programme 1
Programmation, Calcul Symbolique
et Intelligence Artificielle

ON INFINITE LOOPS IN LOGIC PROGRAMMING

Philippe BESNARD

Septembre 1989

RR-1096

IRISA

INSTITUT DE RECHERCHE EN INFORMATIQUE
ET SYSTÈMES ALÉATOIRES

Campus Universitaire de Beaulieu
Avenue du Général Leclerc
35042 - RENNES CÉDEX
FRANCE
Tél. : (99) 36.20.00
Télex : UNIRISA 95 0473 F

On infinite loops in logic programming

Sur les boucles infinies en programmation en logique

Publication Interne n° 488 - Septembre 1989 - 20Pages

Philippe Besnard

IRISA
Campus de Beaulieu
35042 Rennes Cédex
FRANCE

Septembre 1989

Abstract: Various techniques for providing logic programming interpreters (and likewise compilers) with a capability to preclude some infinite loops to arise are examined. In the meantime, the most fundamental aspects of the problem of infinite loops in logic programming are illustrated by way of paradigmatic logic programs.

Résumé: Nous examinons un certain nombre de techniques permettant de doter les interpréteurs (ou compilateurs) pour langages de programmation en logique d'une capacité de détection de boucles infinies. Nous donnons dans le même temps quelques programmes logiques à caractère paradigmatique pour les principaux aspects du problème des boucles infinies en programmation en logique.

CENTRE NATIONAL DE LA RECHERCHE SCIENTIFIQUE
(L.A. 227)
UNIVERSITÉ DE RENNES 1 I.N.S.A. DE RENNES

INSTITUT NATIONAL DE RECHERCHE
EN INFORMATIQUE ET EN AUTOMATIQUE
(LABORATOIRE DE RENNES)

On infinite loops in logic programming

Philippe Besnard

IRISA
Campus de Beaulieu
35042 Rennes Cédex
FRANCE

1. Motivation

Ultimately, the idea behind logic programming (and Prolog) is that the algorithms to be executed are just pieces of declarative knowledge in form of logic formulas (Horn clauses as far as Prolog is concerned). This raises the question of the control of the inferences when a logic program is being executed. Control primitives exist, especially in Prolog, which are designed to answer this question. Unfortunately, control primitives destroy the declarative nature of logic programs. It is then important that interpreters (or compilers) of logic programming languages are in full charge of the control, especially in the domain of infinite loops. Indeed, a logic program is intended to be a specification of a problem, so that it should not have anything to do with a notion of looping.

2. Preliminaries

We first recall some basic notions about logic programming. However, the reader is supposed to be familiar with the terminology in logic programming.

A *logic program* is a finite set of clauses (the assertions), each of which has exactly one positive literal. Executing a logic program amounts to resolving, upon the clauses of the program, a negative clause (the query) formed of negative literals (the goals). The resolution method used is SLD-resolution which is the theoretical model for interpreters in logic programming [Apt & van Emden 82] [Lloyd 87]. Given a logic program and a query, SLD-resolution gives a search space which consists of the sequences of SLD-resolvents that are to be computed by an interpreter for the query to be answered.

Exemple 1: Consider the logic program

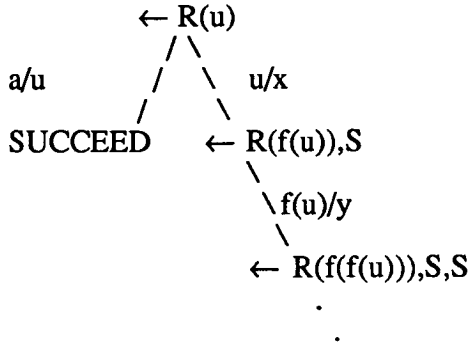
$R(a) \leftarrow$

$R(x) \leftarrow R(f(x)), S$

together with the negative clause

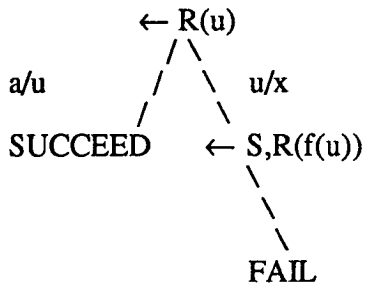
$\leftarrow R(u)$

whose meaning is intuitively "is there a u that satisfies R?". Consider now a standard Prolog interpreter (that is, an interpreter whose selection function for the next literal to be resolved upon is LIFO). The corresponding search space is



This search space is an infinite tree. Hence, the corresponding execution is to be infinite as well because it consists in exploring the search space.

Given a logic program and a query, there may be more than one search space depending on the selection function employed. For our example, there even exists a selection function that yields a finite search space:



In the second resolvent of the rightmost path, the literal to be resolved upon is S. However, S can be unified with no head of a clause in the program. At execution time, the computation is finite and stops after the search tree has been exhaustively explored.

Conventions: We have already employed several implicit conventions, to be used throughout, which are as follows.

-in a resolvent, the literal to be resolved upon is the one immediately following the arrow ←

-t/v is the notation for a substitution in which t is the term to be substituted for the variable v

-a, b, c... (letters from the beginning of alphabet) are constants whereas x, y, z, u... (letters from the end of alphabet) are variables

-R, S... are relations and f, g... are functions

-the variables of a clause in the logic program at hand are renamed whenever the clause is involved in an SLD-resolution.

3. Detecting infinite loops

There exist several approaches to the looping problem in logic programming. First, it can be viewed as a problem to be solved by the programmer [Nute 85] [Poole & Goebel 85]. For the reasons indicated in the introduction, this is methodologically not so viable. A better way of dealing with the looping problem would be to look for a selection function that selects the recursive relations last. A first step in this direction is Mu-Prolog whose primitive "wait" delays the evaluation of a relation [Naish 85].

In order to detect infinite loops, a promising idea is to define a rule indicating whether a finite sequence of SLD-resolvents is the initial part of an infinite path of the search space. This approach is absolutely general: Such a rule is compatible with any selection function. We now review some proposals in this approach.

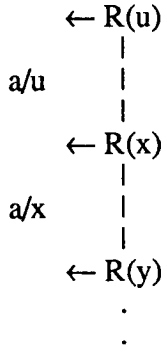
3.1 Repeated application of a clause

Walker [Brough & Walker 84] suggests to preclude infinite sequences of SLD-resolvents by allowing a clause of the logic program to be used at most once.

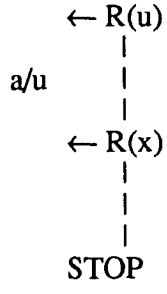
Exemple 2: Consider the logic program consisting of the single clause

$R(a) \leftarrow R(x)$

The search space corresponding to the negative clause $\leftarrow R(u)$ is infinite



Using Walker's rule, one gets a finite search space



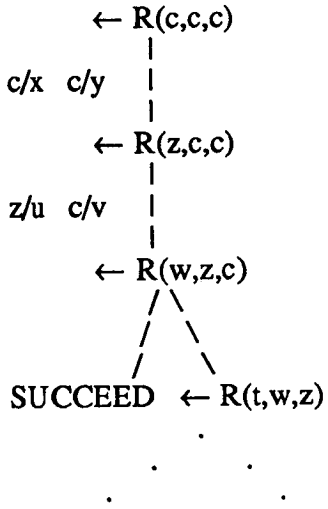
Walker's rule is clearly too rough because it often discards all success paths of a search space. In other words, Walker's rule yields an incomplete system (however, it always terminates).

Exemple 3: Consider the logic program

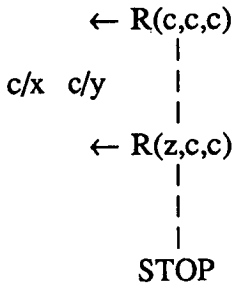
R(a,b,c) ←

R(x,y,c) ← R(x,y,z)

together with the negative clause ← R(c,c,c). The corresponding search space is



Using Walker's rule, one gets a search space with no success path



So, Walker's rule yields an incorrect behaviour: The answer provided to the query "does R(c,c,c) hold?" is "no" instead of "yes". Walker's rule is in fact excessive and one should look for another rule.

3.2 Repeated goal

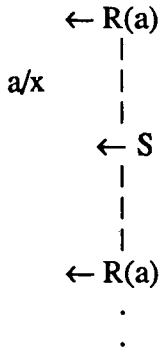
Brough [Brough & Walker 84] proposes to prune a path in which a goal is its own subgoal (regardless of whether this results from direct recursion or not).

Exemple 4: Consider the logic program

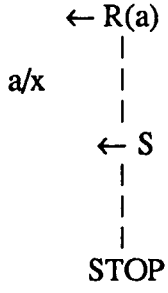
R(x) ← S

S ← R(a)

together with the negative clause ← R(a). The corresponding search space is



Using Brough's rule, one gets a finite search space



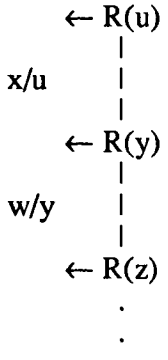
The recursive occurrence of the literal R(a) is thus prevented and the query "is R satisfied by a?" does not give rise to an infinite computation.

As regards termination, Brough's rule is not very satisfactory because it does not prune so many infinite paths.

Exemple 5: Consider the logic program

R(x) ← R(y)

together with the negative clause ← R(u). The corresponding search space is infinite, even if Brough's rule is employed:

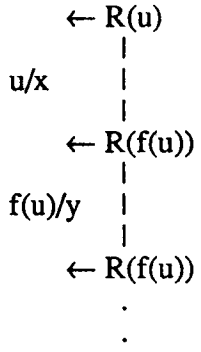


One can say that Brough's rule has no effect in general in the case of SLD-resolutions involving clauses having some variables occurring only in the positive literal.

Exemple 6: Consider the logic program

$R(x) \leftarrow R(f(x))$

together with the negative clause $\leftarrow R(u)$. The corresponding search space is infinite and unaffected by Brough's rule:



In fact, Brough's rule is of no use in the presence of infinite loops involving functions that contribute to build more and more complex terms.

3.3 Repeated goal through syntactic variants

[Covington 85] states a rule for detecting infinite loops arising from biconditional, symmetric or transitive relations. The biconditional relations are of the form

$R(x) \leftarrow S(x)$

$S(x) \leftarrow R(x)$

The symmetric relations are of the form

$R(x,y) \leftarrow R(y,x)$

Covington's idea is to test for a repeated goal in two SLD-resolvents with exactly one SLD-resolvent in between. The transitive relations are of the form

$R(x,z) \leftarrow R(x,y), R(y,z)$

In this case, a loop is detected because R(x,z) has R(x,y) as an immediate subgoal and because R(x,z) and R(x,y) can be unified by means of a substitution involving variables only.

The scope of Covington's rule is rather limited because biconditionality, symmetry and transitivity can all be expressed in an indirect manner.

Exemple 7: The following logic program exhibits a case of indirect biconditionality

$R(x,y) \leftarrow S(x,y)$

$S(x,y) \leftarrow T(x,y)$

$T(x,y) \leftarrow R(x,y)$

Of course, the test for detecting the repeated goals can be done over all pairs of SLD-resolvents. In this way, the problem of indirect recursion is solved but, even though optimisations of the number of tests are possible, the original simplicity of Covington's rule is definitively lost (but [van Gelder 87] is an interesting optimization). Besides, Covington's rule destroys the completeness of SLD-resolution.

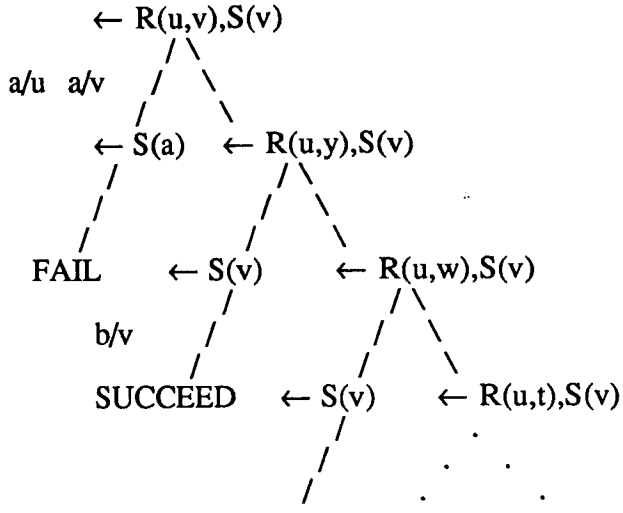
Exemple 8: Consider the logic program

$R(x,z) \leftarrow R(x,y)$

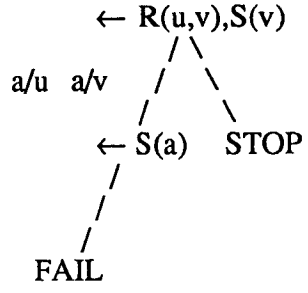
$R(a,a) \leftarrow$

$S(b) \leftarrow$

together with the negative clause $\leftarrow R(a,b), S(b)$. The corresponding search space is



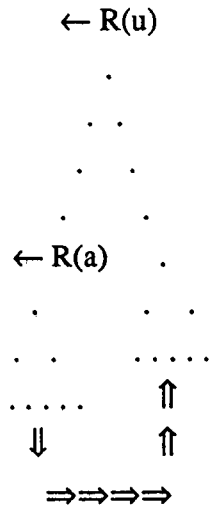
Using Covington's rule, one gets a search space with no success path



4. A contextual approach

In order to solve a problem, one never need to solve in the meanwhile a subproblem harder than the initial problem. As regards logic programming, the notion of goal can be identified with that of problem and subsumption can be seen as a way of furnishing a notion of difficulty. Hence, an approach to looking for finite search spaces in logic programming might be to detect when a goal subsumes one of its subgoals.

Exemple 9: Consider the following search space



In order to solve R(u), one obviously does not need to solve R(a). The clauses that would enable one to solve R(a) could be used to solve R(u) directly.

Let us return to example 8 and look more closely at the point where the SLD-resolvent ←R(u,v),S(v) yields the SLD-resolvent ←R(u,y),S(v). The goal R(u,y) is subsumed by the goal R(u,v) but the SLD-resolvent ←R(u,y),S(v) is not subsumed by ←R(u,v),S(v). In fact, R(u,y) has been recursively introduced while a link between variables has been destroyed (hence, a constraint has been suppressed). As a consequence, the SLD-resolvent ←R(u,y),S(v) is easier to solve than ←R(u,v),S(v). All this means that a subgoal, if subsumed, should be pruned from the search space only if no link between variables has been destroyed.

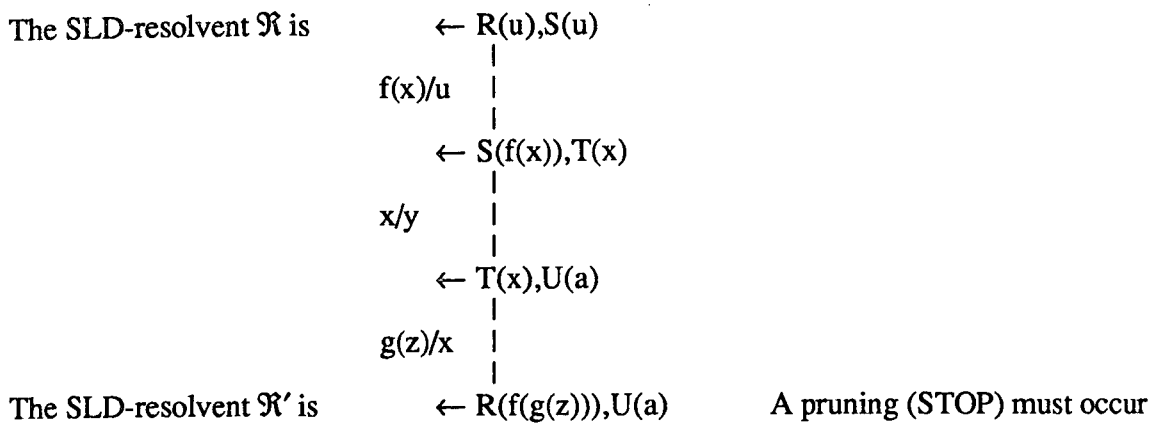
Let us formulate our rule of detection as follows:

Loop detection test
Let $\mathfrak{R}$ be an SLD-resolvent containing a goal G.
Let $\mathfrak{R}'$ be an SLD-resolvent containing G', a subgoal (not necessarily an immediate one) of G.
Let Θ be the composition of the substitutions involved in the SLD-resolution from $\mathfrak{R}$ to $\mathfrak{R}'$.
A pruning must occur at $\mathfrak{R}'$ (denoted by STOP) if *substituting* G' *for* G *in* $\mathfrak{R}\Theta$ *yields an instance of* $\mathfrak{R}$.

Exemple 10: Consider the logic program

R(f(x)) ← T(x)
S(f(x)) ← U(a)
T(g(x)) ← R(f(g(x)))
U(x) ←

together with the negative clause ← R(u), S(u). The corresponding search space is

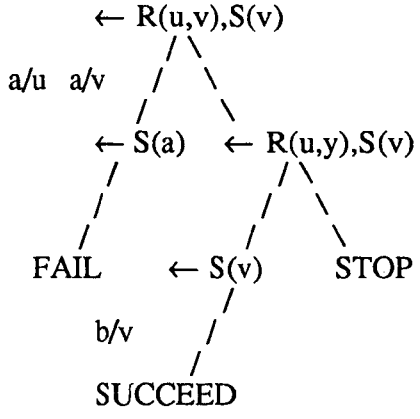


Clearly,
-the goal G is R(u)
-the goal G′ is R(f(g(z))
-the substitution Θ has f(g(z))/u as its most important component
-the clause ←R(f(g(z))),S(f(g(z))) resulting from replacing G by G′ in ℜΘ is indeed an instance of ℜ.

ℜ does not really subsumes ℜ′. Rather, ℜ subsumes the underlying current SLD-resolvent.

Our method is about detecting repeated goals, in a way that extends Brough's and Covington. Roughly speaking, the principle is as follows. An SLD-resolvent is confronted with a variant obtained by replacing the repeated goal. The criterion is merely substitutional but it can detect SLD-resolvents subsumptions.

Using our criterion for loop detection, example 8 is then dealt with in a fully satisfactory manner, as follows.



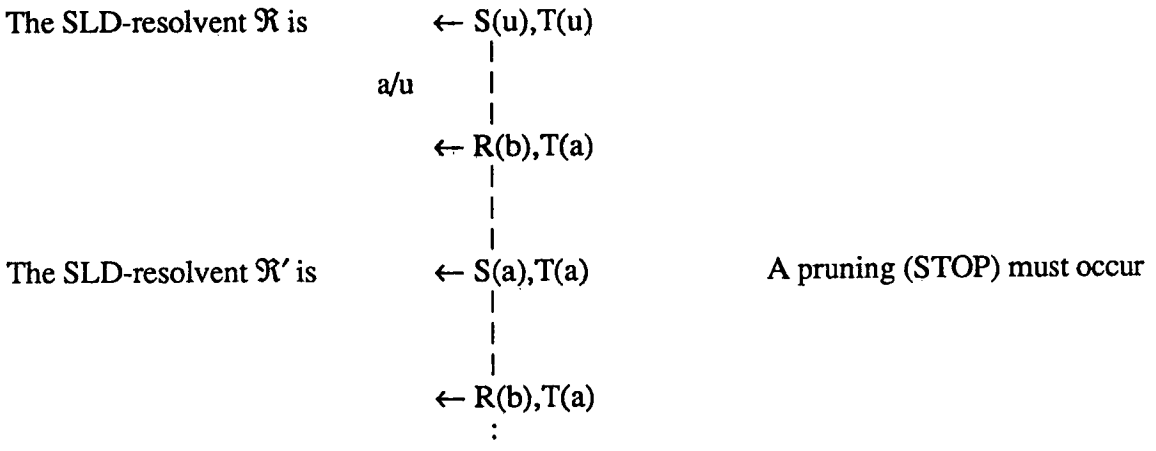
The criterion used also detects the loops where a link between variables has been restaured after it had been broken. This is illustrated by the next example.

Exemple 11: Consider the logic program

$R(x) \leftarrow S(a)$

$S(a) \leftarrow R(b)$

together with the negative clause $\leftarrow S(u), T(u)$. The corresponding search space is



Our method preserves the completeness of SLD-resolution, as stated by Theorem 1. Of course, by a restricted SLD-tree we mean the search space resulting from pruning according to our criterion the standard search space for SLD-resolution.

Theorem 1: Any SLD-tree with a success path contains a restricted SLD-tree with a success path.

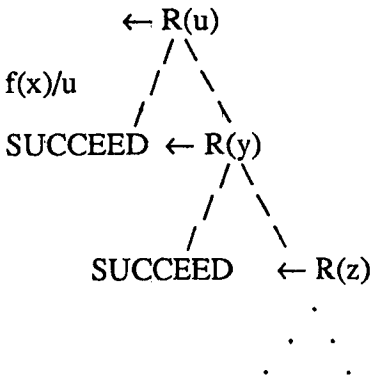
Theorem 1 indeed expresses that our criterion never prunes all the success paths of a search space.

Exemple 12: Consider the logic program

$R(f(x)) \leftarrow R(y)$

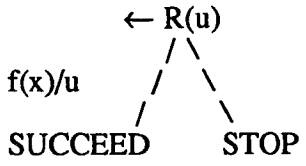
$R(f(x)) \leftarrow$

together with the negative clause $\leftarrow R(u)$. The corresponding search space is



There is an infinite number of solutions, all of them are syntactic variants of the generic solution u=f(x)

Our criterion allows us to restrain the search space so that only the generic solution is kept.



Theorem 2: Let P be a logic program with no function and such that all variables occurring in the body of a clause also occur in the head of the clause. Then all restricted SLD-trees for P are finite.

The interest of Theorem 2 is illustrated by the next example, knowing that our loop detection method preserves the completeness of SLD-resolution (as proved by Theorem 1).

Example 13: Consider the logic program P defined by the two clauses

$R(x,y) \leftarrow R(z,x), R(z,y)$

$R(a,b) \leftarrow$

Consider now the logic program P′

$R(x,y) \leftarrow R(a,x), R(a,y)$

$R(x,y) \leftarrow R(b,x), R(b,y)$

$R(a,b) \leftarrow$

obtained from P by replacing each clause by its instance where all variables occur in the positive literal. P and P′ are logically equivalent but our loop detection method transforms every infinite search space over P′ into a finite one (provided that the root of the original search space is a negative clause with no function).

Example 13 suggests how to get a restriction of SLD-resolution which is decidable for datalog programs (no function in the logic programs and in the queries).

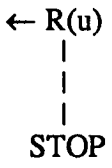
5. Open problems

There remain some serious problems. First, our method may be satisfactory for a logic program P and a query Q even though it is not for the same program P and a query Q′ subsumed by Q. The next example illustrates this paradoxical behaviour.

Exemple 14: Consider the logic program

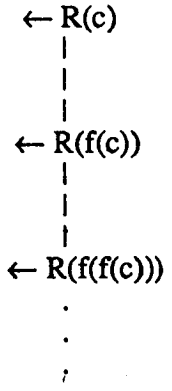
$R(x) \leftarrow R(f(x))$

Let Q be the query "is there a u satisfying R?". The corresponding negative clause $\leftarrow R(u)$ yields the following search space



That is, it can be established that there is no u satisfying R.

Let Q' be the query "does c satisfy R?". The corresponding infinite search space is not pruned:



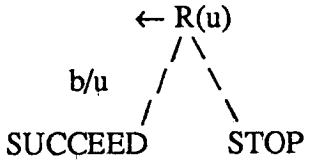
Another problem is that there exist search spaces in which not all solutions are preserved from being pruned.

Exemple 15: Consider the logic program

R(b) ←

R(c) ← R(b)

together with the negative clause ← R(u). Although both b and c are answers to the query "is there a u satisfying R?", only b appears in the search space



Notice that there would be a success path in the search spaces attached to ←R(b) and ←R(c) as well as ←R(u),DIF(u,b) (provided that the search space corresponding to DIF(c,b) contains a success path -the case where u is bound to b is discarded by DIF(b,b) being not satisfied-).

Apart from the problems evoked by example 14 and example 15, a major problem concerns devising an extension of our method for dealing with negation. One knows that there are tracherous difficulties with negation in logic programming when variables are involved. As regards loop checking, even the propositional case is tracherous when logic programs that are not stratifiable are considered.

Exemple 16: Consider the (general) logic program

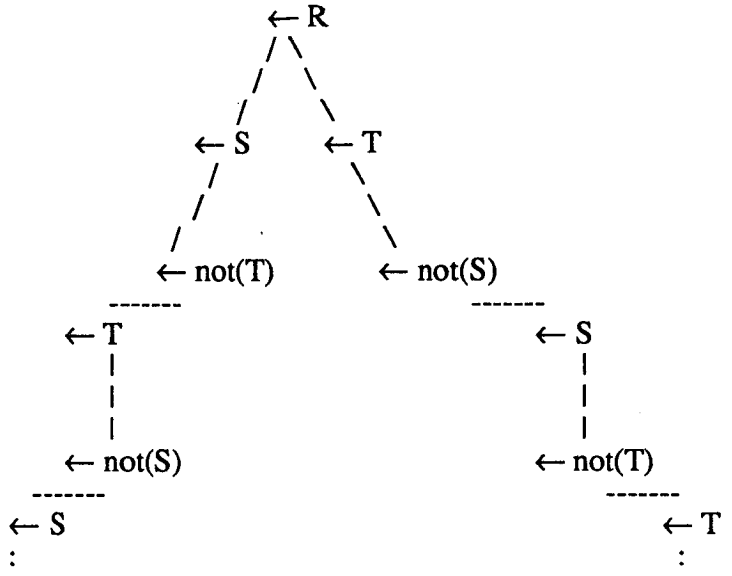
$R \leftarrow S$

$R \leftarrow T$

$S \leftarrow not(T)$

$T \leftarrow not(S)$

This represents the logical theory $\{S \rightarrow R, T \rightarrow R, S \vee T, T \vee S\}$ whose R is a logical consequence. Despite, the search space for $\leftarrow R$ does not contain any success path



Since there is no success path in the original search space, what would it mean to prune it at $\leftarrow S$ and $\leftarrow T$ so as to get a finitely failed search space?

6. Conclusion

The problem of loop detection in logic programming exhibits different levels of difficulty depending for instance on the presence of functional terms or negation in the logic programs. The paradigm for the simplest problem of loop detection (no negation and no functional term) is given by example 17.

Exemple 17: Consider the family of logic programs defined by

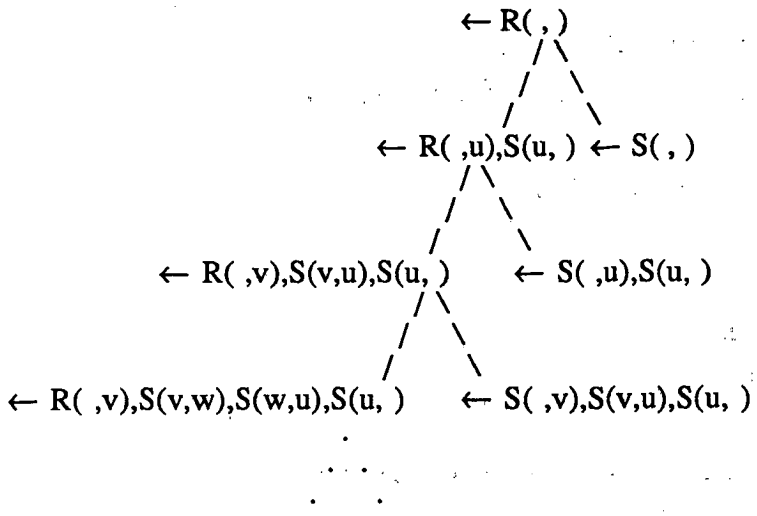
$R(x,z) \leftarrow R(x,y), S(y,z)$

$R(x,y) \leftarrow S(x,y)$

supplemented with a certain amount of ground clauses of the form

$S(\alpha,\beta) \leftarrow$

Then, the search space attached to the negative clause $\leftarrow R(\ ,\ )$ is



In order to preserve the completeness of SLD-resolution over this family of cases, the leftmost path should be developed to a depth that can only be determined by an exhaustive examination of the clauses that define S. That is, there exist search strategies that are not compatible with a pruning technique strong enough to yield completeness and decidability (over datalog programs) of the resulting refinement of SLD-resolution: In the example, a depth-first strategy for exploring the search space either leads to incompleteness or does not lead to a decision procedure. Indeed, the only way to prune correctly the leftmost path is by taking into account the search space for the relation S (either directly from the logic program or indirectly by using for example a breadth-first strategy for exploring the search space). In effect, taking into account the search space for the basic cases of recursive relations is known as an effective method for controlling recursive inference [Smith & al. 86]. As a matter of fact, [Tamaki & Sato 86] explicitly describe such a procedure tailored to logic programming. However, for techniques that can unproblematically be combined with any strategy for exploring the search space, a systematic investigation is due to [Apt & al. 89] (for this particular subject [Bol & al. 89] is even more illuminating).

7. References

Apt K. R., Bol R. N. & Klop J. W. [1989]

On the safe termination of Prolog programs.

Proc. 6th Conf. on Logic Programming, Levi G. & Martelli M. (eds.), MIT Press, Cambridge MA

Apt K. R. & van Emden M. H. [1982]

Contribution to the theory of logic programming.

Jour. of the Assoc. for Computing Machinery 29 (3), pp. 841-862.

Bol R. N., Apt K. R. & Klop J. W. [1989]

An analysis of loop checking mechanisms for logic programs.

Technical report, Centre for Mathematics and Computer Science, Amsterdam.