OLD Resolution with Tabulation

Hisao TAMAKI
Ibaraki University
Nakanarusawa 4-12-1
Hitachi, 316, JAPAN

Taisuke SATO
Electrotechnical Laboratory
Umezono 1-1-4
Sakuramura, 305, JAPAN

ABSTRACT

To resolve the search-incompleteness of depth-first logic program interpreters, a new interpretation method based on the tabulation technique is developed and modeled as a refinement to SLD resolution. Its search space completeness is proved, and a complete search strategy consisting of iterated stages of depth-first search is presented. It is also proved that for programs defining finite relations only, the method under an arbitrary search strategy is terminating and complete.

1. Introduction

The most fundamental principle of logic programming is the equivalence of the declarative and the procedural semantics[1], which has led many researchers to believe logic programming to be a suitable framework for various program manipulation tasks such as verification, synthesis, transformation, and formal debugging. The unfortunate fact is that this equivalence is always sacrificed in real implementations like Prolog, for the sake of execution efficiency. As a result, logically correct logic programs do not necessarily run correctly on Prolog.

One may argue that Prolog is just a programming language with its own procedural semantics, of which programmers should have sufficient knoledge. But the abandonment of the equivalence is so deeply concerned with the philosophy and the potential of logic programming that it could not be approved easily.

There are several causes of this dis-equivalence: the absence of occur check, the depth-first search strategy, and inclusion of many extra-logical feartures. We atack the second problem in this paper: we develope an interpretation method which is complete even under essentially depth-first search strategies.

The completeness of SLD refutation [2,3,4] ensures that given a conjunction of atomic formulas as a query, every instance of it implied by the program can be obtained as a result of some computation path. Though a typical Prolog interpreters are essentially SLD refutation procedures, they are not complete in the sense that they can, with their depth-first search strategy, be trapped by an infinite computation path in the search tree, and fail to find an actually existing successful computation path.

The breadth-first strategy might seem a sufficient theoretical answer to this problem, since it will eventually find any successful computation path in the search tree. But apart from the practical problem of its storage requirement, it also suffers from infinite paths of the search tree. In this case, though it is not trapped from succesful computation paths, it can be trapped from termination if all solutions are requested, even when the set of solutions is finite.

Several authors, including Brough and Walker [5], proposed tecniques to prune the infinite paths in the search tree by detecting identical or matching goals on a path. However, all of such tecniques are incomplete in two ways (as was studied in [5]) : some infinite paths can escape the pruning, and some pruned infinite paths can have side branches which constitute successfull computaion paths.

The interpretation method we develope here can be considered as a remedy to such pruning tecniques, as well as a generalization of the tabulation tecniques [6] for functional programs, where the result of evaluating a function call is stored in a table to eliminate repeated evaluation of the function calls for the same arguments. Though the same effect of avoiding the redundant evaluation of a goal is achieved as a side bennefit, the principal purpose of applying the technique here is to prevent the interpreter from repeatedly entering the evaluation of the same goal in a single computation path and thus from being trapped by an infinite path. The suspended computation node in the path is later fed, through the table, with the solutions of the other computation paths for the goal in question. In this way, the completeness of the SLD refutation is preserved.

The next Section presents an example to illustrate the above discussion and to informally explain our interpretation method. Section 3 and Section 4 contain more detailed description and some completeness results. Finally in Section 5, we conclude by summarizing the advantage of our method.

2. Examples

2.1 Infinite paths in search trees

Consider the following program to define the reachability relation in a directed graph. (We follow the DEC10 Prolog convention of designating variables by upper case letters.)

PROGRAM 2.1 (graph reachability)

```
(C1)   reach(X,Y) ← reach(X,Z), edge(Z,Y).
(C2)   reach(X,X).
(C3)   edge(a,b).
(C4)   edge(a,c).
(C5)   edge(b,a).
(C6)   edge(b,d).
```

Fig.2.1 shows the search tree, which we call an *OLD tree*, for a query ←*reach*(*a*,*X*) given to this program. Each node is (labeled with) a *goal statement* or *negative clause*, and each child node of a node is the result of applying a clause in the program to the *leftmost* goal (atom) of the parent goal statement. The symbol □ denotes the *empty goal statement* or *null clause*. Each edge is labeled with the substitution for the variables of the parent goal statement necessary to make this clause application possible. An OLD tree is a special case of an SLD tree[3,4].

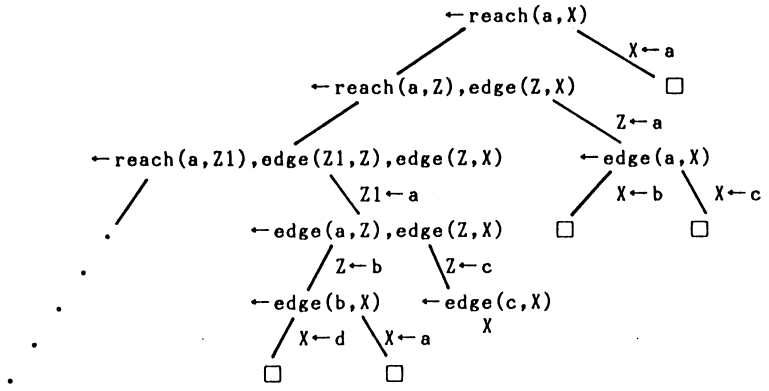

Fig.2.1 an OLD tree

We can learn several things from this example.

(1) The depth-first interpreter is trapped by the leftmost infinite path giving no solution, provided the two clauses for *reach* is ordered as in the program.

(2) If the order of the clauses for *reach* is reversed, the interpreter runs infinitely repeating the solutions.

(3) The behavior of (2) is related to the nature of the graph in question. If it is acyclic, the interpreter gives the finite set of solutions and then goes into the infinite path without giving any further solutions.

(4) The infiniteness of the search tree is partly due to the left-recursive style of the definition of the predicate *reach*. Though the right-recursive reformulation of the program succeeds in eliminating the infiniteness if the object graph is acyclic, it does not work for graphs with cycles like this example.

In this simple example, an experienced programmer could immediately modify the program, adopting right-recursion and an explicit data structure for the set of already reached nodes, to run it correctly on Prolog. But there are cases where such programming solutions are neither trivial nor preferable, just as the conversion from left-recursive grammars to right-recursive ones is not always preferable. In fact, this work is motivated by a desire to run directly a dataflow analyzer of logic programs, which is concisely formulated by definite clause logic but loops on

ordinary prolog interpreters. The programming effort to avoid such looping is certainly what we would like to dispense with, at least in the early stage of the developement.

2.2. Illustration of the method

Now we return to the first example to show how our interpretation method works. We start with the root of the search tree, labeled with the goal ←*reach*(*a*,*X*), which is stored in a table called the *solution table*, to be associated with the list of its solutions. Expansion by the clause (C1) gives the new goal statement ←*reach*(*a*,*Z*),*edge*(*Z*,*X*), and expansion by the unit clause (C2) gives the null statement with the solution *reach*(*a*,*a*), which is stored into the solution list of *reach*(*a*,*X*) (Fig.2.2).

Since the subgoal *reach*(*a*,*Z*) is an instance of the goal *reach*(*a*,*X*) in the solution table, expansion of the node 2 is suspended and the reference to the solution list is established via another table called the *lookup table*.

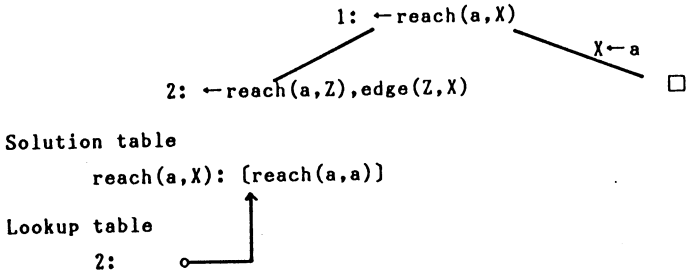

Fig.2.2

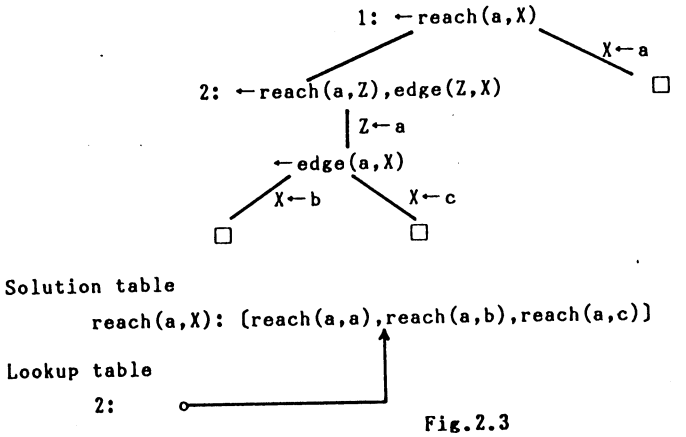

Fig.2.3

Unlike the pruning techniques mentioned in Section 1, we do not stop here but use the solution in the solution list to expand the goal statement of the node 2, obtaining two more solutions to the goal on the root (Fig.2.3). The pointer in the

lookup table makes this *solution lookup* possible, and is now advanced to point to the list of new solutions.

Application of the second and the third solutions gives only one more solution (Fig.2.4). Then finally the application of the fourth solution adds nothing to the solution list, and the whole process terminates (Fig.2.5).

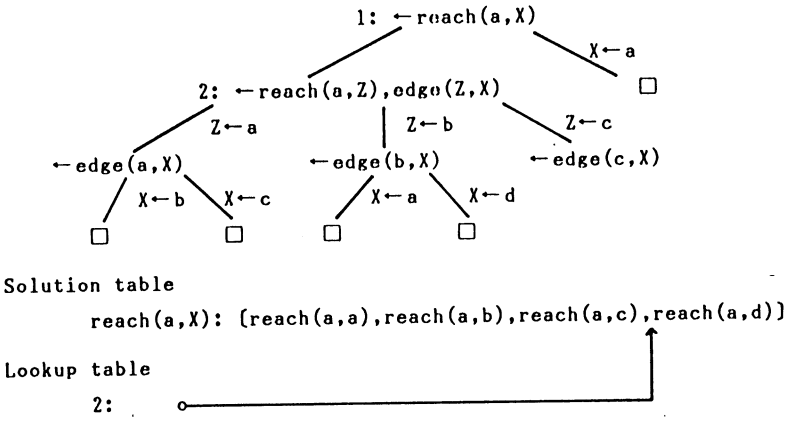

Fig.2.4

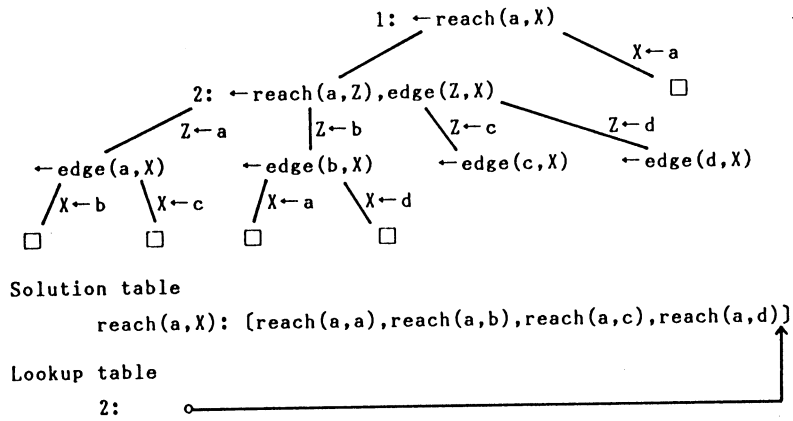

Fig.2.5

This example might be too simple to illuminate the inportant fact that the top down process (ordinary expansion) and the bottom up process (table lookup) can arbitrarily be intermingled with each other. For example, it is possible that some goal statement resulted from a table lookup is executed further in top down manner, and then suspended again as another entry in the lookup table. The detailed description and the completeness proof in the succeeding sections are thus motivated.

It should also be noted that the entire process is controlled by the top down search, and only those solutions required by the top goal are generated. For example, if we add arbitrary number of nodes and edges not reachable form 'a' to the graph in this example, the interpretation process described above remains completely unchanged.

3. OLDT refutation and its completeness

In this section, we formulate the interpretation method as OLDT refutation and prove its completeness. Mostly we follow the standard terminology and conventions in the basic theory of logic programming [3,4], and omit definitions of basic terms and notions, such as *term*, *atom*, *definite clause*, *negative clause*, *null clause*, *unification*, and so on.

First we model the conventional Prolog interpreter by means of *OLD refutation*, which is a special case of SLD refutation[3].

DEFINITION 3.1 (OLD resolution)

Let C be a negative clause $\leftarrow A_1,..,A_n$ $(n>0)$ and D be a definite clause. Let D', of the form $A \leftarrow B_1,..,B_m$ $(m \geq 0)$, be D with all variables renamed so that there is no conflict with those in C. C and D are said to be *OLD resolvable* if A_1 and A are unifiable, and the negative clause (or null clause when n = 1 and m = 0) $\leftarrow (B_1,..,B_m,A_2,..,A_n)\theta$ is the *OLD resolvent* of C and D where θ is the mgu of A_1 and A. The restriction of the substitution θ to the variables of A_1 is called *the substitution of the OLD resolution*.

The OLD resolvent and the substitution of the resolution are unique up to renaming of variables.

DEFINITION 3.2 (OLD tree)

Let P be a program and C_0 be a negative clause. Then the *OLD tree* for the pair (P, C_0) is a possibly infinite tree with its nodes labeled with negative or null clauses so that the following condition is satisfied.

1. The root is labeled with C_0.
2. Assume a node v is labeled with C.
   2.1 If C is a null clause then v is a terminal node.
   2.2 Otherwise, let $D_1,..,D_n$ $(n \geq 0)$ be all the clauses in P which are OLD resolvable with C, and $C_1,..,C_n$ the respective OLD resolvents. Then v has n child nodes, labeled with $C_1,..,C_n$. The edge from v to the node labeled with C_i is labeled with θ_i, where θ_i is the substitution of the OLD resolution of C and Di.

DEFINITION 3.3 (OLD refutation)

Given a program P and a negative clause C, an *OLD refutation* of C by P is a path in the OLD tree of (P, C), from the root to a node labeled with the null clause. Let $\theta_1,\ldots,\theta_k$ be the labeles of the edges on the path. The *substitution of the refutation* is the composition $\theta = \theta_1 \cdot \theta_2 \cdot \ldots \cdot \theta_k$, and the *solution of the refutation* is $C\theta$.

DEFINITION 3.4 (Counterexample of a negative clause)

Given a program P, an instance $\leftarrow A_1,\ldots,A_n$ of a negative clause C is said to be a *counterexample* of C in P if the universal closure of $A_1\&..\&A_n$ is a logical consequence of P.

The soundness and completeness of OLD refutation is just an instance of those of general SLD refutation [2,3,4].

THEOREM 3.5 (Soundness of OLD refutation)

If C' is the solution of an OLD refutation of a negative clause C by P, C' is a counterexample of C in P.

THEOREM 3.6 (Completeness of OLD refutation)

If C' is a counterexample of a negative clause C in P, there is an OLD refutation of C by P such that C' is an instance of the solution of the refutation.

The following notion of subrefutation is specific to the OLD refutation and will be frequently used in the sequel.

DEFINITION 3.7 (Subrefutation)

For a node v in an OLD tree, we denote the number of atoms in the negative clause labeling v by $leng(v)$.

Consider a path from a node v_1 in an OLD tree to one of its descendants v_2 such that for every node v on the path other than v_2 $leng(v) > leng(v_2)$ holds. Let $\leftarrow A_1,\ldots,A_n$ be the label of v_1, where $n = leng(v_1)$, and let $k = n - leng(v_2)$. Since this path can be viewed as a refutation of $\leftarrow A_1,\ldots,A_k$ by neglecting last $leng(v_2)$ atoms in the label of every node on the path, we call it a *subrefutation* of $\leftarrow A_1,\ldots,A_k$. We call it a *unit subrefutation* if $k = 1$. The substitution and the solution of a subrefutation is defined as for a refutation.

We need a few more definitions before describing the OLDT refutation.

DEFINITION 3.8 (Partial OLD tree)

A partial OLD tree is a finite top segment of an OLD tree. That is, any finite tree obtained by deleting arbitrary number of subtrees from an OLD tree is a partial OLD tree.

DEFINITION 3.9 (Table predicates and their term-depth)

We assume that among the predicate symbols used in the program, some are designated by the programmer as *table predicates*. Moreover, we assume that each table predicate p is assigned a non-negative integer called the *term-depth* of p.

The intention of the first assumption is to give flexibility to our method by limiting the tabulation only to the designated table predicates. As an extreme, the OLD refutation will be a special case of OLDT refutation, where no predicates are designated as table predicates. The second assumption is related to the abstraction operation defined below, which is used to bound the number of distinct subgoals (atoms) to be solved.

DEFINITION 3.10 (Term-depth abstraction)

Let A be an atom of a table predicate p, and k be the term-depth of the p. Then the term-depth abstraction of A, denoted by $abs(A)$ is A, with every subterm of depth more than k replaced by distinct new variables.

For example, if the term-depth of p is 1, abs(p(f(g(X),h(Y)),a)) is p(f(U,V),a).

DEFINITION 3.11 (OLDT structure)

An *OLDT structure* is a forest of partial OLD trees with two tables, the *solution table* and the *lookup table*.

A node is called a *table node* if the leftmost atom of its label is of a table predicate. A table node is either a *lookup node* or a *solution node*. The solution table associates the leftmost atom of the label of each solution node with a list of instances of that atom, called the *solution list*. The lookup table associates each lookup node with a pointer pointing *into* some solution list in the solution table.

We now describe the valid construction process of OLDT structures for a given pair of a program and a negative clause.

DEFINITION 3.12 (Table node registration)

Given an OLDT structure and a table node v in it, the *table node registration procedure* classifies it as a solution node or a lookup node, and does necessary table manipulation, resulting in a new OLDT structure.

According to the leftmost atom A of v's label, we distinguish among the following cases. (Note that by definition the predicate of A is a table predicate.)

(1) (Lookup node)
A is an instance of some key entry A' in the solution table.
Put v into the lookup table with the pointer to the entire solution list of A'.

(2) (Abstraction)
Otherwise, and the nesting depth of A is greater than the term-depth of the predicate of A.
Create a new root v0 in the forest, label it with $abs(A)$,
put $abs(A)$ in the solution table with an empty solution list, and
put v in the lookup table with a pointer to this empty solution list.

(3) (Solution node) Otherwise, put A in the solution table with an empty solution list.

DEFINITION 3.13 (Initial OLDT structure)

Given a program P and a negative clause C0, the *initial OLDT structure* for the pair (P, C_0) is the result of the following operation.

(1) Let T_0 be an OLDT structure consisting of a forest with a single node v_0, labeled with C_0, an empty solution table, and an empty lookup table.

(2) Apply the table node registration procedure to the node v_0 in T_0.

DEFINITION 3.14 (Extension of an OLDT structure)

Given a program P and an OLD structure T, an *immediate extension of T by P* is the result of either of the following operations.

(1) (OLD extension) Select a terminal node v, which is not a lookup node, such that its label C is not a null clause and at least one clause in P is OLD resolvable with C.

(1.1) Let $D_1,..,D_n$ $(n \geq 1)$ be all the clauses in P which are OLD resolvable with C, and $C_1,..,C_n$ the respective OLD resolvents. Then add n child nodes, labeled with $C_1,..,C_n$, to v. The edge from v to the node labeled with each C_i is labeled with θ_i, where θ_i is the substitution of the resolution of C and D_i.

(1.2) For each new node, register it if it is a table node.

(1.3) For each unit subrefutation (if any) starting from a solution node and ending with some of the new nodes, assume that the subrefutation is of $\leftarrow A$, and let $\leftarrow A'$ be its solution. Add A' to the last of the solution list of A, if A' is not an instance of any entry in the solution list.

(2) (Lookup extension) Select a lookup node v, such that the pointer associated with it points to a nonempty sublist of a solution list. Let A be the head element of this sublist. Advance the pointer by one to skip A. Let C be the label of v. If C and $A\leftarrow$ are OLD resolvable, then create a child node of v, labeled with the resolvent, and label the new edge with the substitution of the resolution. Do the same thing as in (1.3).

An OLDT structure T' is an *extension* of another OLDT structure T, if T' is obtained from T through succesive application of immediate extensions.

DEFINITION 3.15 (OLDT refutation)

Given a program P and a negative clause C, an *OLDT refutation* of C by P is a path in some extension of the initial OLDT structure for (P, C), from the *initial root* to a node labeled with the null clause. Here, by initial root we mean the root inherited from the initial OLDT structure.

The notions such as the substitution and the solution of refutation or subrefutation are defined similarly as for OLD refutation.

Note that an OLDT refutation by P is an OLD refutation by P plus some set of unit clause theorems of P for table predicates. Thus the soundness of OLDT refutation is an immediate consequence of the soundness of OLD refutation. For the completeness proof we need the following lemma.

DEFINITION 3.16 (subsumption of subrefutation)

A (sub)refutaion r is said to *subsume* a (sub)refutation r' if the solution of r' is an instance of the solution of r.

LEMMA 3.17 (OLDT simulation of OLD subrefutation)

Assume r is an OLD subrefutation of a negative (or null) clause $\leftarrow A_1,..,A_n$ $(n \geq 0)$ by a program P, T is an OLDT structure for the clause and the program, and v is a node in T. Assume further that v is labeled with $\leftarrow B_1,..,B_m$, $m \geq n$, and the sequence $A_1,..,A_n$ is an instance of $B_1,..,B_n$.

Then there exists an extension of T such that T contains an OLDT subrefutation of $\leftarrow B_1,..,B_n$ which starts from v and subsumes r.

Proof. The proof is by induction on the triple (r,T,v), ordered by the following well-founded ordering.

(r,T,v) precedes (r',T',v')

iff $|r| < |r'|$, or

$|r| = |r'|$ and v' is a lookup node but v is not,

where $|r|$ means the length of the refutation.

Induction basis: $|r| = 1$. Trivial since r is a subrefutation of a null clause.

Induction step: We consider two cases depending on whether the node v is a lookup node or not.

(*Case* 1): v is a lookup node. Then there is a corresponding solution node v' in T. Let r be divided as concatenation of two subrefutations r_1 and r_2 so that r_1 is a subrefutation of $\leftarrow A_1$. Since $|r_1| \leq |r|$, and A_1 is an instance of the leftmost atom B_1 of the label of v, hence of the leftmost atom B_1' of the label of v', by the induction hypothesis we have an extension T' of T such that T' contains a subrefutation of $\leftarrow B_1'$ which starts from v' and subsumes r_1. That is, if we let $\leftarrow A_1'$ be the solution of the subrefutation r_1, and $\leftarrow B_1''$ be the solution of the above OLDT subrefutation, $\leftarrow A_1'$ is an instance of $\leftarrow B_1''$. By the operation (1.3) of the definition of the OLDT structure extension, the solution list of B_1' in T' includes B_1''.

Now consider the negative clause $\leftarrow B_1,..,B_n$ and the unit clause $B_1'' \leftarrow$. Since their instances $\leftarrow A_1,..,A_n$ and $A_1' \leftarrow$ have an OLD resolvent $\leftarrow A_2',..,A_n'$, they also have an OLD resolvent $\leftarrow B_2',..,B_n'$ such that $\leftarrow A_2',..,A_n'$ is an instance of $\leftarrow B_2',..,B_n'$. This means that T' can be extended (if necessary) to T'' by lookup extension, so that the node v has a child node v'' with the first $n-1$ atoms of its label being $B_2',..,B_n'$.

Since r_2 is a subrefutation of $\leftarrow A_2',..,A_n'$ and $|r_2| < |r|$, again by the induction hypothesis we have an extension T''' of T'' which contains an OLDT subrefutation s of $\leftarrow B_2',..,B_n'$, starting from v'' and subsuming r_2. The path in T''' starting from v and followed by the subrefutation s constitutes the required subrefutation of $\leftarrow B_1,B_2,..,B_n$.

(*Case* 2): v is not a lookup node. Let u and r' be the first node and the remaining

path of the subrefutation r, and D be the definite clause in P used in the first step of the subrefutation r. Then r' is a subrefutation of $\leftarrow L_1,\ldots,L_k,A_2',\ldots,A_n'$, the OLD resolvent of $\leftarrow A_1,\ldots,A_n$ and D. By the assumption, the label $\leftarrow B_1,\ldots,B_m$ of v and D are also OLD resolvable, and the resolvent $\leftarrow L_1',\ldots,L_k',B_2',\ldots,B_m'$ is such that the sequence $L_1,\ldots,L_k,A_2',\ldots,A_n'$ is an instance of the sequence $L_1',\ldots,L_k',B_2',\ldots,B_n'$. Extending T (if necessary) by the OLD resolution on the node v, we can get an OLDT structure T' in which v has a child node v' labeled with $\leftarrow L_1',\ldots,L_k',B_2',\ldots,B_m'$. Then by the induction hypothesis we have an extention T'' of T' which contains a subrefutation s of $\leftarrow L_1',\ldots,L_k',B_2',\ldots,B_n'$, starting from v' and subsuming r'. The path in T'' starting from v and followed by s constitutes the required subrefutation of $\leftarrow B_1,B_2,\ldots,B_n$. □

THEOREM 3.18 (Completeness of OLDT refutation)

Let P be a program, C_0 a negative clause. Assume that an instance C_0' of C_0 is a counterexample of C_0 in P. Then any extension of the initial OLDT structure for (P, C_0) can be further extended to contain an OLDT refutation of C_0, such that C_0' is an instance of the solution of the refutation.

Proof. By the completeness of OLD refutation, there exists an OLD refutation satisfying the above condition. Lemma 3.17, applied to this OLD (sub)refutation, the given OLDT structure, and the initial root of the OLDT structure, provides the required extension. □

Before concluding this section, it should be stressed that OLDT structures are conceptual structures like SLD trees for formal treatment, and need not be fully maintained in real implementations. What portion of the structure is to be maintained depends on the search strategy employed, just as in the case of the SLD tree.

4. Search strategies

Search strategies for OLDT refutation determine, at each step of OLDT structure construction, which of the extension operations is to be applied to which node, when there are several possibilities. Unfortunately, not all search strategies are complete. For example, consider the following program.

PROGRAM 4.1

(C1) p(X) ← q(X), r.
(C2) q(s(X)) ← q(X).
(C3) q(0).
(C4) r.

Fig.4.1 is a possible snap shot of an OLDT structure for the query $\leftarrow p(X)$.

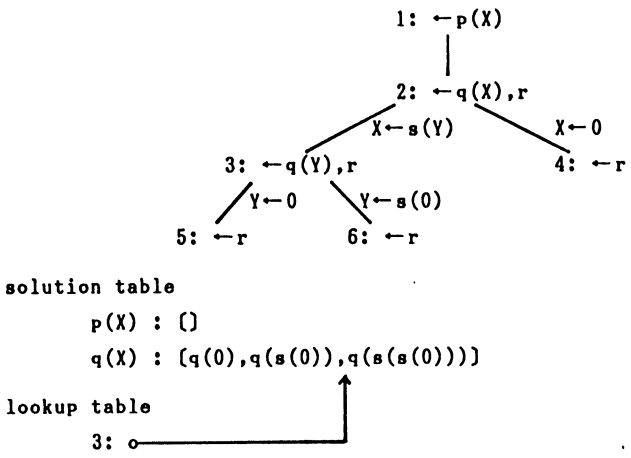

Fig.4.1

If the extension by lookup continues to be preferred, the refutations of the initial goal $\leftarrow p(X)$ will never be found.

It is not difficult to avoid this situation: we have only to prevent sticking to some (group of) lookup node(s). The following strategy is a candidate suitable for sequential implementations.

DEFINITION 4.1 (Multistage depth-first strategy)

We assume here that the forest in an OLDT structure is an *ordered forest*: the roots are ordered by the order of creation, the child nodes of a non-lookup node are ordered by the textual order of clauses used, and the child nodes of a lookup node are again ordered by the order of creation. We say a node u is *to the left of* a node v, if u precede v in the left-to-right post order traversal of the ordered forest.

The search process in the *multistage depth-first strategy* consists of multiple stages. At each step in the i-th stage, one of the following extension operation is applied to the node of the current OLDT structure which is leftmost among the possible.

(1) OLD extension.

(2) Lookup extension, with the solutions to be looked up limited to those *which are generated in the (i-1)th or earlier stages.*

When there are no nodes to which they are applicable, one stage is finished and the next stage begins. When a stage adds no solutions to the solution lists, the entire process terminates.

For example, consider Program 4.1 executed under the multistage depth-first strategy. The first stage ends with the OLDT structure shown in Fig.4.2, successfully generating a solution $p(0)$ to the top goal. The lookup extension of the node 3 is suppressed since the only solution $q(0)$ in the corresponding solution list is generated in this stage. The second stage generates another solution $p(s(0))$ to

the top goal (Fig.4.3). The possibility of non-productive iteration suggested in Fig.4.1 is avoided by prohibiting lookup to the solution generated within the current stage. The succeeding stages generate solutions $p(s(s(0)))$, ..., giving the complete set.

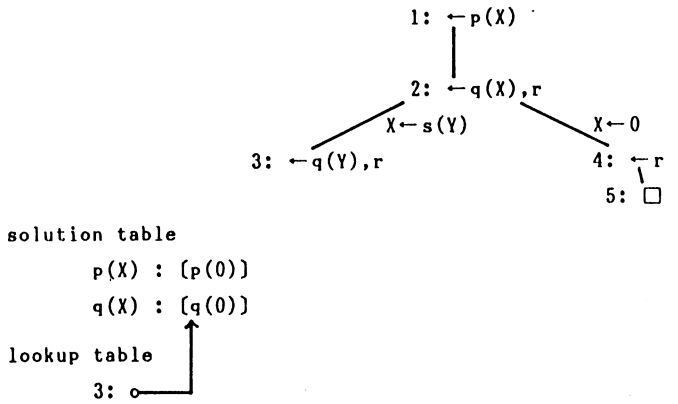

Fig.4.2

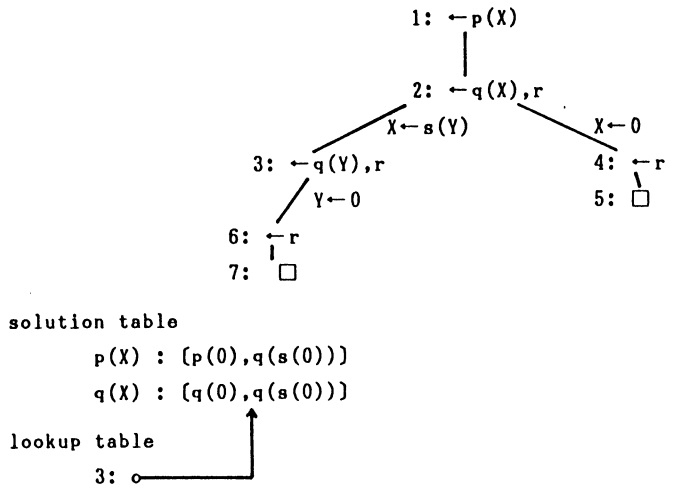

Fig.4.3

Note that the mode of search within a stage is exactly depth-first, with the solution lists treated as additional data bases of assertions. Lookup nodes alone can be resumed for further search in succeeding stages.

THEOREM 4.2 (Completeness of the multistage depth-first strategy)

Assume that all predicates in a program P is designated as table predicates. Then for any negative clause C_0 and its counterexample C_0' in P, the search process for OLDT refutation of (P, C_0) under the multistage depth-first strategy finds a refutation such that C_0' is an instance of its solution.

Proof. Noting that the number of solution nodes in any OLDT structure for the program P is bounded by a constant owing to the abstraction operation in the table node registration procedure, this is an easy consequence of the completeness of OLDT refutation. The details are omitted. □

In the special case where all the relations defined by the program is finite, OLDT refutation has a nice property which OLD refutation lacks. (See the example in Section 2.)

THEOREM 4.3 (Completeness and termination for finite-model programs)

Assume that the minimum Herbrand model of a program P is finite, and all predicates in P is designated as table predicates. Then for any negative clause C_0, the search process for the OLDT refutation for (P, C_0) under *any* search strategy terminates, and gives a complete set of solutions.

Proof. Since the length of a solution list in any OLDT structure for (P, C_0) is bounded by a constant, the branching factor of lookup nodes is bounded by a constant. The length of a path is also bounded by a constant, since the number of solution nodes is bounded by a constant and every lookup node in a path decrease the number of atoms in the label by one. Thus the size of the OLDT structures is bounded by a constant. Therefore the completeness of the search directly follows the completeness of the OLDT refutation. □

REMARK (Comparison with the bottom up interpretation method)

Since this termination property is also possesed by the usual bottom up interpretation method, it should be compared with our method.

The bottom up interpretation also consists of succesive stages. In each stage, every positive unit theorem directly derivable from a definite clause in the program and positive unit theorems obtained in previous stages is calculated. The process terminates when a stage does not produce any new theorems. When the minimum model is finite, it obviously terminates giving the complete set of positive unit theorems.

The advantage of our method over the bottom up interpretation is that it is essentially top down, and only those theorems required by the top goal is derived, in principle. We say 'in principle', because the abstraction operation generalizes a goal and may require solutions which is not required by the original goal. In fact, if we set the term-depth of every predicate to be 0, then the multistage search for the OLDT refutation becomes nothing but an implementation of the bottom up interpretation method.

The abstraction operation is, however, a kind of theoretical safety valve, and it seems that in most applications we can set appropriate term-depth for each predicate

so that the abstraction operation never actually occurs, as in the example of Section 2, preserving the top-down nature of our interpretation method.

5. Conclusion

We do not claim that the interpretation method described above should be a single, ultimate solution to the problem of the search-incompleteness of Prolog: the storage requirement can be too demanding in some cases, and the overhead of table manipulation can be too large.

Rather, the advantage of the method exists in providing a spectrum of procedural approximations to the declarative semantics: as two extremes, if we designate all predicates as table predicates, then the multistage strategy gives the complete interpretation procedure; if we choose no predicates as table predicates, then the multistage search for OLDT refutation is exactly the same as the depth-first search for OLD refutation. For some programs the latter extreme is still an exact approximation, but for others we must choose some appropriate intermediate approximations. The common techniques for proving termination could be used for this purpose, to ensure that some predicates need not be designated as table predicates.

The ideal logic programming system which we envision will consist of a variety of approximating implementation methods, tools to determine which approximation is exact or sufficient, and a powerful set of optimization tecniques, rather than of a complete and efficient universal interpreter.

References

[1] Van Emden, M.H. and Kowalski, R.A. "The semantics of predicate logic as a programming languages", *Journal of the ACM* 23, No.4, 1976.

[2] Clark, K.L. "Predicate logic as a computational formalism", Imperial College research monograph 79/59 TOC, December 1979.

[3] Apt, K.R. and Van Emden, M.H. "Contributions to the theory of logic programming", *Journal of the ACM* 29, NO.3, 1982.

[4] Lloyd, J.W. *Foundations of logic programming*, Springer-Verlag, 1984.

[5] Brough, D.R. and Walker, A. "Some practical properties of logic programming interpreters", *Proc. International Conference on FGCS 1984*, Tokyo, Nov. 1984.

[6] Bird, R.S. "Tabulation techniques for recursive programs", *Computing Surveys* 12, No.4, 1980.