Linear Tabulated Resolution Based on Prolog Control Strategy

Yi-Dong Shen
Department of Computer Science, Chongqing University, Chongqing 400044, P.R.China

Li-Yan Yuan and Jia-Huai You
Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada T6G 2H1

Neng-Fa Zhou
Kyushu Institute of Technology, Iizuka, Fukuoka 820, Japan

Abstract

Infinite loops and redundant computations are long recognized open problems in Prolog. Two ways have been explored to resolve these problems: loop checking and tabling. Loop checking can cut infinite loops, but it can not be both sound and complete even for function-free logic programs. Tabling seems to be an effective way to resolve infinite loops and redundant computations. However, existing tabulated resolutions, such as OLDT-resolution, SLG-resolution and Tabulated SLS-resolution, are all non-linear because they rely on the solution-look up mode in formulating tabling. The principal disadvantages of non-linear resolutions are that they can not be implemented using a simple stack-based memory structure like that in Prolog and some strictly sequential operators such as cuts in Prolog can not be guaranteed to work normally.

In this paper, we propose a hybrid way to resolve infinite loops and redundant computations. We combine the ideas of loop checking and tabling to establish a linear tabulated resolution called TP-resolution. TP-resolution has the following distinctive features: (1) It does not distinguish between solution and lookup nodes; any nodes can resolve table subgoals against program clauses as well as table facts. (2) It makes linear tabulated derivations in the same way as Prolog except that infinite loops are broken and redundant computations are avoided. It deals with cuts as effectively as Prolog. (3) It is sound and complete for positive logic programs with the bounded-term-size property. The underlying algorithm is simple and can be implemented by a slight extension to any existing Prolog abstract machines such as WAM or ATOAM.

1 Introduction

While Prolog has many distinct advantages, it suffers from some serious problems, among the best-known of which are infinite loops and redundant computations. Infinite loops make users (especially less skilled users) lose confidence in writing terminable Prolog programs, whereas redundant computations greatly reduce the efficiency of Prolog. Existing approaches to resolving these problems can be classified into two categories: loop checking and tabling.

Loop checking is a direct way to cut infinite loops. It locates nodes at which SLD-derivations step into a loop and prunes them from SLD-trees. Informally, an SLD-derivation $G_0 \Rightarrow C_1, \theta_1 G_1 \Rightarrow ... \Rightarrow C_i, \theta_i G_i \Rightarrow ... \Rightarrow C_k, \theta_k G_k \Rightarrow ...$ is said to step into a loop at a node $N_k$ labeled with a goal $G_k$ if there is a node $N_i$ (0 ≤ $i$ < $k$) labeled with a goal $G_i$ in the derivation such that $G_i$ and $G_k$ are sufficiently similar. Many loop checking mechanisms have been presented in literature (e.g. [1, 2, 7, 8, 14, 15, 17, 19]). However, no loop checking mechanism can be both (weakly) sound and...
complete because the loop checking problem itself is undecidable in general even for function-free logic programs [1].

The main idea of tabling is that during top-down query evaluation, we store intermediate results of some subgoals and look them up to solve variants of the subgoals that occur later. Since no variant subgoals will be recomputed by applying the same set of program clauses, infinite loops can be avoided. As a result, termination can be guaranteed for bounded-term-size programs and redundant computations substantially reduced [4, 6, 16, 19, 21].

There are many ways to formulate tabling, each leading to a tabulated resolution (e.g. OLDT-resolution [16], SLG-resolution [6], Tabulated SLS-resolution [4], etc.). However, although existing tabulated resolutions differ in one aspect or another, all rely on the so-called solution-lookup mode. That is, all nodes in a search tree/forest are partitioned into two subsets, solution nodes and lookup nodes; solution nodes produce child nodes using program clauses, whereas lookup nodes produce child nodes using answers in tables. For instance, in OLDT-resolution solution nodes are those at which the left-most subgoals are generated earliest among all their variant subgoals [16]. In SLG-resolution solution nodes are roots of trees in a search forest, each labeled by a special clause of the form $A \leftarrow A$ [5]. In Tabulated SLS-resolution, any root of a tree in a forest is itself labeled by an instance, say $A \leftarrow B_1, ..., B_n$ ($n \geq 0$), of a program clause and no nodes in the tree will produce child nodes using program clauses [3]. For any atom $A$ we can assume a virtual super-root labeled with $A \leftarrow A$, which takes all the roots in the forest labeled by $A \leftarrow ...$ as its child nodes. In this sense, the search forest in Tabulated SLS-resolution is the same as that in SLG-resolution for positive logic programs. Therefore, we can consider all virtual super-roots as solution nodes.

Our investigation shows that the principal disadvantage of the solution-lookup mode is that it makes tabulated resolutions non-linear in the sense that their derivations cannot be formulated in a linear form $G_0 \Rightarrow C_1 \theta_1 \ G_1 \Rightarrow \ldots \Rightarrow C_i \theta_i \ G_i \Rightarrow \ldots$, where each $G_{i+1}$ ($i \geq 0$) is derived from $G_i$ by resolving a subgoal $A$ in $G_i$ against a program clause (or an answer in a table) $C_{i+1}$ with an mgu $\theta_{i+1}$ [9]. It is due to such non-linearness that the underlying tabulated resolutions cannot be implemented in the same way as SLD-resolution (Prolog) using a simple stack-based memory structure and some strictly sequential operators such as cuts (!) are no longer guaranteed to work normally. For instance, in the well-known tabulated resolution system XSB clauses like $p(.) \leftarrow ..., t(.), !, ..., w$ with $t(.)$ a table subgoal, are not allowed because the table predicate $t$ occurs in the scope of a cut [11, 13].

The objective of our research is to establish a hybrid approach to resolving infinite loops and redundant computations and develop a linear tabulated Prolog system. In this paper, we establish a theoretical framework for such a system, focusing on a linear tabulated resolution — TP-resolution for positive logic programs (TP for Tabulated Prolog).

Remark: In this paper we will use the prefix $TP$ to name some key concepts such as TP-strategy, TP-tree, TP-derivation and TP-resolution, in contrast to Prolog- (control) strategy, Prolog-tree (i.e. SLD-tree generated under Prolog-strategy), Prolog-derivation and Prolog-resolution (i.e. SLD-resolution controlled by Prolog-strategy), respectively.

In TP-resolution, each node in a search tree can act not only as a solution node but also as a lookup node,

1In fact, we do not distinguish between solution and lookup nodes in TP-resolution.
child nodes ($N_i$ is then as a solution node). The order of using answers in a table is *first-generated-first-use* and the order of selecting program clauses is from top to bottom except for the case where the derivation steps into a loop at $N_i$. In such a case, the subgoal $A$ skips the clause that is being used by its closest ancestor subgoal that is a variant of $A$. Like OLDT-resolution, TP-resolution is sound and complete for positive logic programs with the bounded-term-size property.

The plan of this paper is as follows. In Section 2 we present a typical example to illustrate the main idea of TP-resolution and its key differences from existing tabulated resolutions. In Section 3, we formally define TP-resolution. In Section 3.1 we discuss how to determine table predicates, how to represent tables and how to operate on tables. In Section 3.2 we first introduce the so called PMF mode for resolving table subgoals with program clauses, which lays a basis for a linear tabulated resolution. We then define a tabulated control strategy called *TP-strategy*, which enhances Prolog-strategy with proper policies for controlling table fact selection. Next we present a constructive definition (an algorithm) of a *TP-tree* based on TP-strategy. Finally, based on TP-trees we define *TP-derivations* and *TP-resolution*.

Section 4 is devoted to showing some major characteristics of TP-resolution, including its termination property and soundness and completeness. We also discuss in detail how TP-resolution deals with the cut operator. Finally, in Section 5 we conclude the paper with some further work.

We assume familiarity with the basic concepts of logic programming, as presented in [10]. Here and throughout, variables begin with a capital letter, and predicates, functions and constants with a lower case letter. By $E$ we denote a list/tuple $(E_1, \ldots, E_m)$ of elements. Let $X = (X_1, \ldots, X_m)$ be a list of variables and $I = (I_1, \ldots, I_m)$ a list of terms. By $X/I$ we denote an mgu $\{X_1/I_1, \ldots, X_m/I_m\}$. By $p(.)$ we refer to any atom with the predicate $p$ and by $p(\bar{X})$ to an atom $p(\bar{X})$ that contains the list $\bar{X}$ of variables. For instance, if $p(\bar{X}) = p(W, a, f(Y), Z)$, then $\bar{X} = (W, Y, Z)$. Let $G \leftarrow A_1, \ldots, A_m$ be a goal and $B$ a subgoal. By $G + B$ we denote the goal $\leftarrow A_1, \ldots, A_m, B$. By a *variant* of an atom (resp. a subgoal or a term) $A$ we mean an atom (resp. a subgoal or a term) $A'$ that is the same as $A$ up to variable renaming. Let $V$ be a set of atoms (resp. subgoals or terms) that are variants of each other; then they are called *variant atoms* (resp. *variant subgoals* or *variant terms*). Moreover, unless otherwise stated, by a (logic) program we refer to a positive logic program with a finite set of clauses. Finally like Prolog we divide a logic program into two parts, facts and rules. Rules are definitions of predicates, whereas facts are (variable-free) instances of predicates. For the same predicate, its facts appear before its rules.

## 2 An Illustrative Example

We use the following simple program to illustrate the basic idea of the TP approach. For convenience of presentation, we choose OLDT-resolution [16] for side-by-side comparison (other typical tabulated resolutions, such as SLG-resolution [6] and Tabulated SLS-resolution [4], have similar effects).

\[
P_1: \text{reach}(X, Y) \leftarrow \text{reach}(X, Z), \text{edge}(Z, Y).
\]

\[
P_1: \text{reach}(X, X).
\]

\[
P_1: \text{reach}(X, d).
\]

\[
P_1: \text{edge}(a, b).
\]

\[
P_1: \text{edge}(d, e).
\]

\[
P_1: \text{reach}(X, Y) \leftarrow \text{reach}(X, Z), \text{edge}(Z, Y).
\]

\[
P_1: \text{reach}(X, X).
\]

\[
P_1: \text{reach}(X, d).
\]

\[
P_1: \text{edge}(a, b).
\]

\[
P_1: \text{edge}(d, e).
\]

\[
P_1: \text{reach}(X, Y) \leftarrow \text{reach}(X, Z), \text{edge}(Z, Y).
\]

\[
P_1: \text{reach}(X, X).
\]

\[
P_1: \text{reach}(X, d).
\]

\[
P_1: \text{edge}(a, b).
\]

\[
P_1: \text{edge}(d, e).
\]

\[
P_1: \text{reach}(X, Y) \leftarrow \text{reach}(X, Z), \text{edge}(Z, Y).
\]

\[
P_1: \text{reach}(X, X).
\]

\[
P_1: \text{reach}(X, d).
\]

\[
P_1: \text{edge}(a, b).
\]

\[
P_1: \text{edge}(d, e).
\]

\[
P_1: \text{reach}(X, Y) \leftarrow \text{reach}(X, Z), \text{edge}(Z, Y).
\]

\[
P_1: \text{reach}(X, X).
\]

\[
P_1: \text{reach}(X, d).
\]

\[
P_1: \text{edge}(a, b).
\]

\[
P_1: \text{edge}(d, e).
\]

\[
P_1: \text{reach}(X, Y) \leftarrow \text{reach}(X, Z), \text{edge}(Z, Y).
\]

\[
P_1: \text{reach}(X, X).
\]

\[
P_1: \text{reach}(X, d).
\]

\[
P_1: \text{edge}(a, b).
\]

\[
P_1: \text{edge}(d, e).
\]

\[
P_1: \text{reach}(X, Y) \leftarrow \text{reach}(X, Z), \text{edge}(Z, Y).
\]

\[
P_1: \text{reach}(X, X).
\]

\[
P_1: \text{reach}(X, d).
\]

\[
P_1: \text{edge}(a, b).
\]

\[
P_1: \text{edge}(d, e).
\]

\[
P_1: \text{reach}(X, Y) \leftarrow \text{reach}(X, Z), \text{edge}(Z, Y).
\]

\[
P_1: \text{reach}(X, X).
\]

\[
P_1: \text{reach}(X, d).
\]

\[
P_1: \text{edge}(a, b).
\]

\[
P_1: \text{edge}(d, e).
\]

\[
P_1: \text{reach}(X, Y) \leftarrow \text{reach}(X, Z), \text{edge}(Z, Y).
\]

\[
P_1: \text{reach}(X, X).
\]

\[
P_1: \text{reach}(X, d).
\]

\[
P_1: \text{edge}(a, b).
\]

\[
P_1: \text{edge}(d, e).
\]

\[
P_1: \text{reach}(X, Y) \leftarrow \text{reach}(X, Z), \text{edge}(Z, Y).
\]

\[
P_1: \text{reach}(X, X).
\]

\[
P_1: \text{reach}(X, d).
\]

\[
P_1: \text{edge}(a, b).
\]

\[
P_1: \text{edge}(d, e).
\]

\[
P_1: \text{reach}(X, Y) \leftarrow \text{reach}(X, Z), \text{edge}(Z, Y).
\]

\[
P_1: \text{reach}(X, X).
\]

\[
P_1: \text{reach}(X, d).
\]

\[
P_1: \text{edge}(a, b).
\]

\[
P_1: \text{edge}(d, e).
\]

\[
P_1: \text{reach}(X, Y) \leftarrow \text{reach}(X, Z), \text{edge}(Z, Y).
\]

\[
P_1: \text{reach}(X, X).
\]

\[
P_1: \text{reach}(X, d).
\]

\[
P_1: \text{edge}(a, b).
\]

\[
P_1: \text{edge}(d, e).
\]
Let the query (top goal) be \( \leftarrow \text{reach}(a, X) \). Then Prolog will step into an infinite loop right after the application of the first clause \( C_1 \). We now show how it works using OLDT-resolution (under the depth-first control strategy). Starting from the root node \( N_0 \) labeled with the goal \( \leftarrow \text{reach}(a, X) \), the application of the clause \( C_1 \) gives a child node \( N_1 \) labeled with the goal \( \leftarrow \text{reach}(a, Z), \text{edge}(Z, X) \) (see Fig.1). As the subgoal \( \text{reach}(a, Z) \) is a variant of \( \text{reach}(a, X) \) that occurred earlier, it is suspended to wait for \( \text{reach}(a, X) \) to produce answers. \( N_0 \) and \( N_1 \) (resp. \( \text{reach}(a, X) \) and \( \text{reach}(a, Z) \)) are then called solution and lookup nodes (resp. subgoals), respectively. So the derivation goes back to \( N_0 \) and resolves \( \text{reach}(a, X) \) with the second clause \( C_2 \), which gives a sibling node \( N_2 \) labeled with an empty clause \( \square \). Since \( \text{reach}(a, a) \) is an answer to the subgoal \( \text{reach}(a, X) \), it is memorized in a table, say \( TB(\text{reach}(a, X)) \). The derivation then jumps back to \( N_1 \) and uses the answer \( \text{reach}(a, a) \) in the table to resolve with the lookup subgoal \( \text{reach}(a, Z) \), which gives a new node \( N_3 \) labeled with \( \leftarrow \text{edge}(a, X) \). Next, the node \( N_4 \) labeled with \( \square \) is derived from \( N_3 \) by resolving the subgoal \( \text{edge}(a, X) \) with the clause \( C_4 \). Again the answer \( \text{reach}(a, b) \) is added to the table \( TB(\text{reach}(a, X)) \). After these steps, the OLDT-derivation evolves into a tree as depicted in Fig.1, which is clearly not linear.

![Fig.1 OLDT-derivation.](image)

We now explain how TP-resolution works. Starting from the root node \( N_0 \) labeled with the goal \( \leftarrow \text{reach}(a, X) \) we apply the clause \( C_1 \) to derive a child node \( N_1 \) labeled with the goal \( \leftarrow \text{reach}(a, Z), \text{edge}(Z, X) \) (see Fig.2). As the subgoal \( \text{reach}(a, Z) \) is a variant of \( \text{reach}(a, X) \) and the latter is an ancestor of the former (i.e., the derivation steps into a loop at \( N_1 \) [14]), we choose \( C_2 \), the clause from the backtracking point of the subgoal \( \text{reach}(a, X) \), to resolve with \( \text{reach}(a, Z) \), which gives a child node \( N_2 \) labeled with \( \leftarrow \text{edge}(a, X) \). As \( \text{reach}(a, a) \) is an answer to the subgoal \( \text{reach}(a, Z) \), it is memorized in a table \( TB(\text{reach}(a, X)) \). We then resolve the subgoal \( \text{edge}(a, X) \) against the clause \( C_4 \), which gives the leaf \( N_3 \) labeled with \( \square \). So the answer \( \text{reach}(a, b) \) to the subgoal \( \text{reach}(a, X) \) is added to the table \( TB(\text{reach}(a, X)) \). After these steps, we get a path as shown in Fig.2, which is clearly linear.

![Fig.2 TP-derivation.](image)
Now consider backtracking. Remember that after the above derivation steps, the table $TB(\text{reach}(a, X))$ consists of two answers, $\text{reach}(a, a)$ and $\text{reach}(a, b)$. For the OLDT approach, it first backtracks to $N_3$ and then to $N_1$ (Fig. 1). As the subgoal $\text{reach}(a, Z)$ has used the first answer in the table before, it resolves the second, $\text{reach}(a, b)$, which gives a new node labeled with the goal $\leftarrow \text{edge}(b, X)$. Obviously, this goal will fail, so it backtracks to $N_1$ again. This time no new answers in the table are available to the subgoal $\text{reach}(a, Z)$, so it is suspended and the derivation goes to the solution node $N_0$. The third clause $C_3$ is then selected to resolve with the subgoal $\text{reach}(a, X)$, yielding a new answer $\text{reach}(a, d)$, which is added to the table. The derivation then goes back to $N_1$ where the new answer is used in the same way as described before.

The TP approach does backtracking in the same way as the OLDT approach except for the following key difference: because we do not distinguish between solution and lookup nodes/subgoals, when no new answers in the table are available to the subgoal $\text{reach}(a, Z)$ at $N_1$, we resolve it against the program clause $C_3$. This guarantees that TP-derivations are always linear.

In summary, the OLDT approach makes non-linear tabulated derivations based on the solution-lookup mode, whereas the TP approach makes linear tabulated derivations in which a node/subgoal can act both as a solution node/subgoal and as a lookup node/subgoal.

3 TP-Resolution

This section formally defines the TP approach to tabulated resolution, mainly including the way to determine table predicates, the strategy for controlling tabulated derivations (TP-strategy), and the algorithm for making tabulated derivations based on the control strategy (TP-trees).

3.1 Table Predicates and Tables

In making tabulated derivations, the first issue we encounter is to determine which predicates should be tabled. This is important because if we table too many, the overhead in dealing with the tables will increase, and if we table not enough, we will risk of losing answers (and even going into infinite loops). It turns out, however, that determining precisely which predicates need to be tabled is the same as predicting whether a particular goal will be repeated on a path of an SLD-tree and is undecidable in general. In the well-known tabulated logic programming system XSB [11], there is a procedure named $\text{table\_all}$ (or $\text{auto\_table}$ in its latest version) which automatically chooses predicates to table based on a $\text{predicates\_call\_graph}$. Informally, for any predicates $p$ and $q$, there is an edge $p \rightarrow q$ in a predicates-call graph $PG_P$ if there is a clause in the program $P$ whose head is of the form $p(.)$ and whose body contains a subgoal of the form $q(.)$. Then a predicate $p$ is to be tabled if $PG_P$ contains a cycle with a node $p$.

Choosing table predicates based on a predicates-call graph is safe in the sense that all infinite loops can be avoided by tabling. However, this approach will too easily overtable predicates even for very simple programs like $P = \{ p(a, X) \leftarrow p(b, f(X)) \}$ where it is unnecessary to table the predicate $p$ because neither loops nor redundant computations would occur for any top goals. Although searching for optimal solutions to the problem of choosing table predicates is beyond the scope of the current paper, we suggest using the so-called $\text{atoms\_call\_graph}$ which is defined as follows.

**Definition 3.1** Let $P$ be a logic program and $P'$ the set of clauses in $P$ with non-empty bodies. For each atom $A$ in $P'$, replace each argument of $A$ that is a function by a distinct variable. The
The atoms-call graph of \( P \), denoted \( AG_P \), is a directed graph \(< V_P, E_P >\), where \( V_P \) is the set of nodes and \( E_P \) the set of edges, which is defined inductively as follows.

1. All head atoms (or their variants) of clauses in \( P' \) are in \( V_P \).
2. For each \( A \in V_P \) and clause \( A' \leftarrow B_1, \ldots, B_m \) in \( P' \) with \( A\theta = A'\theta \) (\( \theta \) is an mgu), all \( B_i\theta \) (or their variants) are in \( V_P \) and all \( A \rightarrow B_i\theta \) (or their variants) are in \( E_P \).
3. No two nodes in \( V_P \) are variants.

**Definition 3.2** Let \( P \) be a logic program and \( p \) a predicate in \( P \). \( p \) is a table predicate if \( AG_P \) has a cycle containing a node \( p(\_\_\_\_) \). Any subgoal with a table predicate is called a table subgoal.

**Example 3.1** The atoms-call graph \( AG_{P_1} \) of the program \( P_1 \) in Section 2 is as follows:

\[
\text{reach}(X, Y) \rightarrow \text{edge}(Z, Y)
\]

So only \( \text{reach} \) is a table predicate. For the program \( P = \{ p(a, X) \leftarrow p(b, f(X)) \} \), its atoms-call graph \( AG_P \) is \( p(a, X) \rightarrow p(b, Y) \). We see \( p \) is not a table predicate.

Since an atoms-call graph clearly depicts the backward chaining relationships among atoms, using it to determine table predicates is certainly much more precise than using a predicates-call graph. Moreover, like the predicates-call graph, the atoms-call graph is safe to use (see Lemma 4.2) and easy to build.

With the table predicates in hand, the next issue is how to represent tables for table subgoals. Apparently any table must contain a table subgoal and a set of answers to the subgoal. Note that in our tabling approach, any table subgoal can act both as a solution subgoal and as a lookup subgoal, so a table can be viewed as a blackboard on which a set of variant table subgoals will read and write answers. In order to guarantee not losing answers for any table subgoals (i.e. the table should contain all answers that the table subgoals are supposed to have by applying the related rules), while avoiding redundant computations (i.e. after a rule has been used by a subgoal \( A \), it would not be used by any other subgoal \( A' \) that is a variant of \( A \)), the third component is needed in the table that keeps the status of the rules related to the table predicate. Therefore, after a rule \( R_i \) has been used by a table subgoal \( A \), we change the status of \( R_i \) in the table of \( A \). Then when evaluating a new table subgoal \( A' \) that is a variant of \( A \), the rule \( R_i \) will be excluded. We say that a rule \( R_i \) has been used by a subgoal \( A \) if after the application of \( R_i \), \( A \) tried alternative rules (through backtracking); \( R_i \) has never been used by \( A \) if \( A \) never tried to use it; otherwise \( R_i \) is being used by \( A \). This leads to the following.

**Definition 3.3** Let \( P \) be a logic program and \( p(\bar{X}) \) a table subgoal. Let \( P \) contain exactly \( N_p \) rules, \( R_1, \ldots, R_{N_p} \), with a head \( p(\_\_\_\_) \). A table for \( p(\bar{X}) \) (and all its variants), denoted \( TB(p(\bar{X})) \), is a triple \( (p(\bar{X}), T, R[N_p]) \), where

1. \( T \) is a set of tuples that are instances of \( \bar{X} \), each \( \bar{I} \) of which represents an answer, \( p(\bar{X})\bar{X}/\bar{I} \), to the subgoal, and
2. $R[N_p]$ is an array. Each element $R[i]$ is $-1$, $0$ or $1$, representing that the status of $R_i$ is has been used, is being used and has never been used, respectively, by (any) variant subgoals of $p(X)$.

For convenience, we use $TB(p(X))[i]$ to refer to the $i$-th component of $TB(p(X))$ ($i = 1, 2, 3$), $TB(p(X))[2][j]$ to refer to the $j$-th tuple of the table, and $TB(p(X))[3][j]$ to refer to the status of $R_j$ w.r.t. $p(X)$.

**Example 3.2** Let $P$ be a logic program that contains exactly three rules, $R_1$, $R_2$ and $R_3$, with a head $p(.)$. The table

$TB(p(X,Y)) : (p(X,Y), \{(a,b), (b,a), (b,c)\}, \{-1,0,1\})$

represents that there are three answers to variant subgoals of $p(X,Y)$, namely $p(a,b)$, $p(b,a)$ and $p(b,c)$, and that $R_1$ has already been used by a variant subgoal of $p(X,Y)$, $R_2$ is being used by a variant subgoal of $p(X,Y)$, and $R_3$ has not yet been tried by any variant subgoal of $p(X,Y)$. The table

$TB(p(a,b)) : (p(a,b), \{(\), \{-1,1,1\})$

represents that $p(a,b)$ has been proved true after applying $R_1$ (since $p(a,b)$ contains no variables, the answer is a 0-ary tuple). Finally, the table

$TB(p(a,X)) : (p(a,X), \{\}, \{-1,-1,-1\})$

represents that $p(a,X)$ has no answer at all because $TB(p(a,X))[2]$ remains empty after all the three rules have been applied.

As we mentioned above, a table will be shared by a set of variant subgoals. So our next issue is how each variant subgoal independently operates on a table (including operations like create, read, write, and update). This involves two problems: subgoal identification and answer/clause selection. For the first problem, note that in a set of variant subgoals, some of them may be the same (e.g. let the set be $\{p(X), p(Y), p(X)\}$). In order to uniquely identify these subgoals, we associate with each subgoal a unique node name, say $N_i$, in a tree\(^3\). As a result, a subgoal $A$ at node $N_i$ is different from the subgoal $A$ at node $N_j$ ($i \neq j$), although both are the same atom. For the second problem, note that when evaluating a table subgoal $A$ at node $N_i$, we will apply to it both answers in the table $TB(A)$ and clauses in the program $P$, one by one from top to bottom. So in order to keep track of such answer/clause selection, we attach to $N_i$ two pointers, $PT(N_i)[1]$ that points to an answer in $TB(A)$ (i.e. a tuple in $TB(A)[2]$) and $PT(N_i)[2]$ that points to a clause (rule) in $P$. This leads to the following.

**Definition 3.4** Let $G_i$ be a goal $\leftarrow A_1, \ldots, A_m$ ($m \geq 1$). By register a node $N_i$ with $G_i$ we do the following: (1) label $N_i$ with $G_i$ and associate $N_i$ with $A_1$; (2) create two pointers for $N_i$, $PT(N_i)[1]$ and $PT(N_i)[2]$, which unless otherwise specified are all initialized to null.

Note that since $N_i$ is unique, through it we can uniquely determine a goal $G_i$, subgoal $A_1$, table $TB(A_1)$ and pointers $PT(N_i)[1]$ and $PT(N_i)[2]$.

**Definition 3.5** Let $P$ be a logic program with $M$ rules with a head $p(.)$ and $p(\overline{X})$ be a table subgoal with which the node $N_i$ is associated. We have the following basic operations on a table.

\(^3\)Assume that nodes in a tree have distinct names.
1. **create**\((N_i, p(\bar{X}))\). It creates a table \((p(\bar{X}), T, R_i[1])\) and lets \(PT(N_i)[1]\) point to the first tuple of \(T\) and \(PT(N_i)[2]\) to the first rule in \(P\) with a head \(p(.)\). Here
   
   (a) \(T\) consists of the distinct tuples derived from all facts in \(P\) that are instances of \(p(\bar{X})\), i.e. \(\bar{I} \in T\) if there is a fact \(F\) in \(P\) such that \(F = p(\bar{X})\bar{X}/\bar{I}\);
   
   (b) \(R_i[j] = 1\) for all \(1 \leq j \leq M\).

2. **memo**\((N_i, \bar{I})\), where \(\bar{I}\) is an instance of \(\bar{X}\). It adds \(\bar{I}\) to the end of \(TB(p(\bar{X}))[2]\) if \(\bar{I}\) is not yet in the table.

3. **lookup**\((N_i, \bar{X}, I)\), where \(I\) is a variable. If \(PT(N_i)[1]\) is not null, it binds \(I\) to the tuple pointed by \(PT(N_i)[1]\) and then advances the pointer by 1; otherwise \(I = \text{null}\).

4. **update**\((N_i, J, D)\), where \(1 \leq J \leq M\) and \(D \in \{-1, 0, 1\}\). It sets \(TB(p(\bar{X}))[3][J]\) to \(D\).

First, as we will see in the next subsection, the procedure **create**\((N_i, p(\bar{X}))\) is called only when the subgoal \(p(\bar{X})\) occurs the first time and no variant subgoals occurred before. Therefore, up to the time when we call **create**\((N_i, p(\bar{X}))\), no rules with a head \(p(.)\) in \(P\) have been selected by any variant subgoals of \(p(\bar{X})\), so the status of all the rules should be set to 1 (i.e. all \(R_i[j]\) is 1). Moreover, in order to improve efficiency, we extract onto the answer list \(T\) all facts in \(P\) that are instances of \(p(\bar{X})\), so that all variant subgoals of \(p(\bar{X})\) can fetch those answers directly from the table. Second, whenever a new answer \(p(\bar{I})\) to \(p(\bar{X})\) is derived, we call the procedure **memo**\((N_i, \bar{I})\), which appends \(\bar{I}\) to the end of the answer list in the table. Third, when an answer from a table is requested, we call the procedure **lookup**\((N_i, \bar{X}, I)\), which fetches into \(I\) the tuple pointed by \(PT(N_i)[1]\) and then advances the pointer by 1 to skip the used tuple. As we will see in the next subsection (Definition 3.6), the parameter \(\bar{X}\) in **lookup**\((N_i, \bar{X}, I)\) will be used to form an mgu \(\bar{X}/\bar{I}\). Finally, when the status of rule \(R_i\) is changed to \(D\) by the current subgoal \(p(\bar{X})\), we call the procedure **update**\((N_i, J, D)\) to record the change in the table.

### 3.2 TP-Strategy and TP-Trees

In this subsection, we introduce the tabulated control strategy and the way to make tabulated derivations based on this strategy. We begin by discussing how to resolve subgoals with program clauses and answers in tables.

Let \(N_i\) be a node labeled by a goal \(G_i = A_1, ..., A_m\ (m \geq 1)\) with \(A_1 = p(\bar{X})\) being a table subgoal. Consider evaluating \(A_1\) using a program clause \(C = A \leftarrow B_1, ..., B_n\ (n \geq 0)\), where \(A_1 \theta = A \theta^4\). If we use SLD-resolution, we would obtain a new node labeled with the goal \(G_{i+1} = (B_1, ..., B_n, A_2, ..., A_m)\theta\), where we see that the mgu \(\theta\) is consumed by all \(A_j\ (j > 1)\), although the proof of \(A_1 \theta\) has not yet been completed (produced). In order to avoid such kind of pre-consumption, we propose a so called PMF (for Prove-Memorize-Fetch) mode for resolving table subgoals with clauses. That is, we first prove \((B_1, ..., B_n)\theta\). If it is true with some mgu \(\theta_1\), which means \(A_1 \theta \theta_1\) is true, we memorize the answer \(A_1 \theta \theta_1\) in the table \(TB(A_1)\) if it is new. We then fetch an (new) answer from \(TB(A_1)\) to apply to the remaining subgoals of \(G_i\). Obviously the PMF mode preserves the original set of answers to \(A_1\). Moreover, since only new answers to \(A_1\) are added to the table, all repeated answers to \(A_1\) will be precluded to apply to the remaining subgoals of \(G_i\).

---

4 Here and throughout, we assume that \(C\) has been standardized apart to share no variables with \(G_i\).
The PMF mode can readily be realized by using the two table procedures \texttt{memo(\ldots)} and \texttt{lookup(\ldots)}\footnote{\texttt{memo(\ldots)} and \texttt{lookup(\ldots)} will be reserved to use only for such a purpose in this paper.}. That is, after resolving the subgoal \( A_1 \) with the clause \( C \), \( N_i \) gets a child node \( N_{i+1} \) labeled with the goal \( G_{i+1} \leftarrow (B_1, \ldots, B_n, \theta, \texttt{memo}(N_i, \bar{X} \theta), \texttt{lookup}(N_i, \bar{X}, I_i), A_2, \ldots, A_m) \). Note that the application of \( \theta \) is blocked by the subgoal \( \texttt{lookup}(N_i, \bar{X}, I_i) \) because the consumption (fetch) must be after the production (prove and memorize). We now explain how it works.

Assume that after some resolution steps from \( N_{i+1} \) we reach a node \( N_k \) that is labeled by the goal \( G_k \leftarrow \texttt{memo}(N_i, \bar{X} \theta_1), \texttt{lookup}(N_i, \bar{X}, I_i), A_2, \ldots, A_m \). This means that \( (B_1, \ldots, B_n) \theta \) has been proved true with the mgu \( \theta_1 \). That is, \( A_1 \theta \) is an answer to \( A_1 \). Under the left-most computation rule, \( \texttt{memo}(N_i, \bar{X} \theta_1) \) is executed, which adds the tuple \( \bar{X} \theta_1 \) to the table \( TB(A_1) \) if it is new. Then \( \texttt{lookup}(N_i, \bar{X}, I_i) \) is executed, which fetches into \( I_i \) a tuple from \( TB(A_1) \). As \( A_1 \bar{X}/I_i \) is an answer to the subgoal \( A_1 \) of \( G_i \), the mgu \( \bar{X}/I_i \) needs to be applied to the remaining \( A_j \)'s of \( G_i \). We distinguish between two cases. (1) Among \( A_2 - A_{m-1} \), \( A_j \) is the left-most subgoal of the form \( \texttt{memo}(N_j, \bar{X}) \). So \( A_{j+1} \) must be \( \texttt{lookup}(N_j, \bar{X}, I_i) \). According to the PMF mode, there must be a node \( N_j \), which occurred earlier than \( N_i \), labeled with a goal \( G_j = B, A_{j+2}, \ldots, A_m \) such that \( B \) is a table subgoal and \( \texttt{memo}(N_j, \bar{X}) \) and \( \texttt{lookup}(N_j, \bar{X}, I_i) \) were resulted from resolving \( B \) with a program clause. This means that the proof of \( B \) is now reduced to the proof of \( (A_2, \ldots, A_{j+1}) \bar{X}/I_i \). Therefore, by the PMF mode the application of \( \bar{X}/I_i \) should be blocked at the subgoal \( \texttt{lookup}(N_j, \bar{X}, I_i) \). That is, \( N_k \) has a child node \( N_{k+1} \) labeled with a goal \( G_{k+1} \leftarrow (A_2, \ldots, A_{j+1}) \bar{X}/I_i, A_{j+2}, \ldots, A_m \). (2) No \( A_j \) \((2 \leq j < m)\) is of the form \( \texttt{memo}(\ldots) \). This means that no \( A_j \) is a descendant of some table subgoals, so the mgu \( \bar{X}/I_i \) should be applied to all the \( A_j \)'s. That is, \( G_{k+1} \leftarrow (A_2, \ldots, A_m) \bar{X}/I_i \).

It should be pointed out that by Definition 3.5 the vector \( \bar{X} \) of variables in \( \texttt{lookup}(N_i, \bar{X}, I_i) \) is merely used to form an mgu \( \bar{X}/I_i \) after \( I_i \) is bound to an answer (tuple) in a table, so it can not be instantiated during the resolution.

The above discussion shows how to resolve the table subgoal \( A_1 \) against a program clause. We now consider how to resolve \( A_1 \) with an answer \( I \) in the table \( TB(A_1) \) and how to resolve \( A_1 \) with a program clause \( C \) when \( A_1 \) is a non-table subgoal. Apparently, the first case can be dealt with in the same way as \( \texttt{lookup}(N_i, \bar{X}, I) \). For the second case, as there will be no table for \( A_1 \), we resolve \( G_i \) and \( C \) in the same way as in SLD-resolution except that (by the PMF mode) the application of the mgu \( \theta \) is blocked at the left-most subgoal of the form \( \texttt{lookup(\ldots)} \). In summary, we have the following definition.

**Definition 3.6** Let \( N_i \) be a node labeled by a goal \( G_i \leftarrow A_1, \ldots, A_m \) \((m \geq 1)\).

1. If \( A_1 \) is \( \texttt{memo}(N_h, \bar{I}) \) and \( A_2 \) is \( \texttt{lookup}(N_h, \bar{X}, I_h) \), then (after executing the two table procedures) the \texttt{resolvent} of \( G_i \) and \( I_h \) \((I_h \neq \text{null})\) is the goal \( G_{i+1} \leftarrow (A_3, \ldots, A_k) \bar{X}/I_h, A_{k+1}, \ldots, A_m \), where \( A_{k+1} \) \((3 \leq k)\) is the left-most subgoal of the form \( \texttt{lookup(\ldots)} \).

   Otherwise, let \( A_1 = p(\bar{X}) \) and \( C \) be a program clause \( A \leftarrow B_1, \ldots, B_n \) with \( A \theta = A_1 \theta \).

2. If \( A_1 \) is a non-table subgoal, the \texttt{resolvent} of \( G_i \) and \( C \) is the goal \( G_{i+1} \leftarrow (B_1, \ldots, B_n, A_2, \ldots, A_k) \theta, A_{k+1}, \ldots, A_m \), where \( A_{k+1} \) \((2 \leq k)\) is the left-most subgoal of the form \( \texttt{lookup(\ldots)} \).

3. If \( A_1 \) is a table subgoal, the \texttt{resolvent} of \( G_i \) and \( C \) is the goal \( G_{i+1} \leftarrow (B_1, \ldots, B_n) \theta, \texttt{memo}(N_i, \bar{X} \theta), \texttt{lookup}(N_i, \bar{X}, I_i), A_2, \ldots, A_m \).

4. If \( A_1 \) is a table subgoal, let \( I \) \((I \neq \text{null})\) be an answer (tuple) in the table \( TB(A_1) \), then the \texttt{resolvent} of \( G_i \) and \( I \) is the goal \( G_{i+1} \leftarrow (A_2, \ldots, A_k) \bar{X}/I, A_{k+1}, \ldots, A_m \), where \( A_{k+1} \) \((2 \leq k)\) is the left-most subgoal of the form \( \texttt{lookup(\ldots)} \).
We now discuss tabulated control strategies. Recall that Prolog implements SLD-resolution by sequentially searching an SLD-tree using the Prolog control strategy (Prolog-strategy, for short): **Depth-first** (for goal selection) + **Left-most** (for subgoal selection) + **Top-down** (for clause selection) + **Last-first** (for backtracking). Let “register a node \( N_i \) with \( G_i \)” be as defined by Definition 3.4 with the pointer \( PT(N_i)[1] \) removed. Then the way that Prolog makes SLD-derivations based on Prolog-strategy can be formulated as follows.

**Definition 3.7 (Algorithm 1)** Let \( P \) be a logic program and \( G_0 \) a top goal with the list \( \bar{Y} \) of variables. Let \( \text{return}(X) \) be a procedure that returns \( X \). The Prolog-tree \( T_{G_0} \) of \( P \cup \{G_0 + \text{return}(\bar{Y})\} \) is constructed by recursively performing the following steps until the answer \( NO \) is returned.

1. (Root node) Register the root \( N_0 \) with \( G_0 + \text{return}(\bar{Y}) \) and goto 2.

2. (Node expansion) Let \( N_i \) be the latest registered node labeled by \( G_i = \leftarrow A_1, \ldots, A_m \) \((i \geq 0, m > 0)\). Register \( N_{i+1} \) as a child of \( N_i \) with \( G_{i+1} \) if \( G_{i+1} \) can be obtained as follows.
   - Case 1: \( A_1 \) is \( \text{return}(). \) Execute the procedure \( \text{return}() \), set \( G_{i+1} = \Box \) (an empty clause), and goto 3 with \( N = N_i \).
   - Case 2: \( A_1 \) is an atom. Get a program clause \( A \leftarrow B_1, \ldots, B_n \) (top-down via the pointer \( PT(N_i)[2] \)) such that \( A \theta = A \theta \). If no such a clause exists, then goto 3 with \( N = N_i \); else set \( G_{i+1} = (B_1, \ldots, B_n, A_2, \ldots, A_m) \theta \) and goto 2.

3. (Backtracking) If \( N \) is the root, then return \( NO \); else take its parent node as the latest registered node and goto 2.

Let \( ST_{G_0} \) be the SLD-tree of \( P \cup \{G_0 + \text{return}(\bar{Y})\} \) via the left-most computation rule\(^6\). It is easy to prove that when \( P \) has the bounded-term-size property \([18]\) and \( ST_{G_0} \) contains no infinite loops, Algorithm 1 is sound and complete in that \( T_{G_0} = ST_{G_0} \). Moreover, Algorithm 1 has the following distinct advantages: (1) since SLD-resolution is linear, Algorithm 1 can be efficiently implemented using a simple stack-based memory structure; (2) due to its linearity and regular sequentiality, some useful control mechanisms, such as the well-known cut operator \(!\), can be used to heuristically reduce search space. Unfortunately, Algorithm 1 suffers from two serious problems. One is that it is easy to get into infinite loops even for very simple programs such as \( P = \{p(X) \leftarrow p(X)\} \), which makes it incomplete in many cases. The second problem is that it unnecessarily re-applies the same set of rules to variant subgoals such as in the query \( \leftarrow p(X), p(Y) \), which leads to unacceptable performance.

As tabling has a distinct advantage of resolving infinite loops and redundant derivations, one interesting question then arises: can we enhance Algorithm 1 with tabling, making it free from infinite loops and redundant computations while preserving the above two advantages? In the rest of this subsection, we give a constructive answer to this question. We first discuss how to enhance Prolog-strategy with tabling.

Observe that in a tabulated system, we will have both program clauses and tables. For convenience, we refer to tuples in tables as **table facts**. Therefore, in addition to the existing policies in Prolog-strategy, we need to have the following two additional policies: (1) when both program clauses and table facts are available, first use table facts (i.e. **Table-first** for program and table selection); (2) when there are more than one table fact available, first use that one that is memorized.

\(^6\)In \([16]\), it is called an **OLD-tree**.
earliest. Since we always add new answers to the end of tables (see Definition 3.5 for \textit{memo}(.,)),
policy (2) amounts to saying \textbf{Top-down} selection for table facts. This leads to the following control
strategy for tabulated derivations.

\textbf{Definition 3.8} By \textit{TP-strategy} we mean: Depth-first (for goal selection) + Left-most (for subgoal
selection) + Table-first (for program and table selection) + Top-down (for the selection of table
facts and program clauses) + Last-first (for backtracking).

Our goal is to extend Algorithm 1 to make linear tabulated derivations based on TP-strategy.
To this end, we need to review a few concepts concerning loop checking.

\textbf{Definition 3.9 ([14] with slight modification)} An \textit{ancestor list} \(AL_A\) is associated with each
table subgoal \(A\) in a tree (see the TP-tree below), which is defined recursively as follows.

1. If \(A\) is at the root, then \(AL_A = \{\}\).
2. If \(A\) inherits a subgoal \(A'\) (by copying or instantiation) from its parent node, then \(AL_A =\)
\(AL_{A'}\).
3. Let \(A\) be in the resolvent of a subgoal \(B\) against a rule \(B' \leftarrow A_1, ..., A_n\) with \(B\theta = B'I\theta\)
(i.e. \(A = A_i\theta\) for some \(1 \leq i \leq n\)). If \(B\) is a table subgoal, \(AL_A = AL_B \cup \{B\}\); otherwise
\(AL_A = \{\}\).

We see that for any table subgoals \(A\) and \(A'\), if \(A\) is in the ancestor list of \(A'\), i.e. \(A \in AL_{A'}\), the
proof of \(A\) needs the proof of \(A'\). Particularly, if \(A \in AL_{A'}\) and \(A'\) is a variant of \(A\), the derivation
goes into a loop. This leads to the following.

\textbf{Definition 3.10} Let \(G_i\) and \(G_k\) be two goals in a tabulated derivation and \(A\) and \(A'\) be the left-
most subgoals of \(G_i\) and \(G_k\), respectively. We say \(A\) is an \textit{ancestor subgoal} of \(A'\) if \(A \in AL_{A'}\). If \(A\)
is both an ancestor subgoal and a variant of \(A'\), we say the derivation goes into a \textit{loop}.

We are now in a position to define the TP-tree, which is constructed based on TP-strategy
using the following algorithm.

\textbf{Definition 3.11 (Algorithm 2)} Let \(P\) be a logic program and \(G_0\) a top goal with the list \(\bar{Y}\)
of variables. Let \textit{return} \((\bar{Z})\) be a procedure that returns \(\bar{Z}^T\). The \textit{TP-tree} \(TP_{G_0}\) of
\(P \cup \{G_0 + \text{return}(\bar{Y})\}\) is constructed by recursively performing the following steps until the answer \(NO\) is returned.

1. (Root node) Register the root \(N_0\) with \(G_0 + \text{return}(\bar{Y})\) and goto 2.
2. (Node expansion) Let \(N_i\) be the latest registered node labeled by \(G_i = \leftarrow A_1, ..., A_m\) \((i \geq\)
\(0, m > 0)\). Register \(N_{i+1}\) as a child of \(N_i\) with \(G_{i+1}\) if \(G_{i+1}\) can be obtained as follows.

\begin{itemize}
  \item Case 1: \(A_1\) is \textit{return}(.). Execute the procedure \textit{return}(.), set \(G_{i+1} = \square\) (an empty
clause), and goto 3 with \(N = N_i\).
  \item Case 2: \(A_1\) is \textit{memo}(\(N_h, \bar{I}\)). Then \(A_2\) must be \textit{lookup}(\(N_h, \bar{X}, I_h\)). Execute the two
procedures sequentially. If \(I_h = null\), then goto 3 with \(N = N_i\); else set \(G_{i+1}\) to the
resolvent of \(G_i\) and \(I_h\) and goto 2.
\end{itemize}

\footnote{When \(\bar{Z} = ()\), \textit{return}(\bar{Z})\) returns \textit{YES}.}
Case 3: $A_1$ is a non-table subgoal. Get a program clause $C$ (top-down via the pointer $PT(N_i)[2]$) whose head is unifiable with $A_1$. If no such a clause exists, then goto 3 with $N = N_i$; else set $G_{i+1}$ to the resolvent of $G_i$ and $C$ and goto 2.

Case 4: $A_1 = p(\bar{X})$ is a table subgoal. If no table exists for $A_1$, execute the procedure $create(N_i,A_1)$. Get an instance $I$ of $\bar{X}$ from the table $TB(A_1)$ (top-down via the pointer $PT(N_i)[1]$). If $I \neq null$, then set $G_{i+1}$ to the resolvent of $G_i$ and $I$ and goto 2; else

- Case 4.1: $A_1$ has no ancestor subgoal that is its variant. Get a rule $R_j$ from $P$ (top-down via the pointer $PT(N_i)[2]$) whose head is unifiable with $A_1$ such that $TB(A_1)[3][j] \neq -1$. If no such a rule exists, then goto 3 with $N = N_i$; else set $G_{i+1}$ to the resolvent of $G_i$ and $R_j$, set $TB(A_1)[3][j]$ to 0 and goto 2.

- Case 4.2: Let $N_h$ be the closest node to $N_i$ whose left-most subgoal $A'_1$ is both an ancestor subgoal and a variant of $A_1$. Let the rule being used by $A'_1$ be $R_j$. Get a rule $R_k$ ($k > j$) from $P$ (top-down via the pointer $PT(N_i)[2]$) whose head is unifiable with $A_1$ such that $TB(A_1)[3][k] \neq -1$. If no such a rule exists, then goto 3 with $N = N_i$; else set $G_{i+1}$ to the resolvent of $G_i$ and $R_k$, set $TB(A_1)[3][k]$ to 0 and goto 2.

3. (Backtracking) If $N$ is the root, return NO. Let $N_f$ be the parent node of $N$ with the left-most subgoal $A_{f1}$. If $A_{f1}$ is $memo(_\ldots)$, goto 3 with $N = N_f$. If $A_{f1}$ is a table subgoal and $N$ was generated from $N_f$ by resolving $A_{f1}$ with a rule $R_j$, set $TB(A_{f1})[3][j]$ to $-1$. Take $N_f$ as the latest registered node and goto 2.

Apparently, Algorithm 2 reduces to Algorithm 1 when $P$ contains no table predicates. We now explain Algorithm 2 briefly. First we set up the root $N_0$ via registration (see Definition 3.4). Then by the Depth-first policy we select the latest registered node, say $N_i$ labeled with the goal $G_i$, for expansion (step 2). If the left-most subgoal $A_1$ of $G_i$ is $return(A)$ (Case 1), which means the top goal $G_0$ has been proved true with the answer mgu $\bar{Y}/A$, we reach a success leaf $N_{i+1}$ labeled with $\Box$. We then backtrack (step 3) to derive alternative answers to $G_0$. If $A_1$ is $memo(N_h,\bar{I})$ (Case 2), which means that the left-most subgoal $A_{h1}$ at node $N_h$ is proved true with the answer mgu $\bar{X}/\bar{I}$ ($\bar{X}$ is the list of variables in $A_{h1}$), we memorize $\bar{I}$ in the table $TB(A_{h1})$. Since $A_2$ is $lookup(N_h,\bar{X},I_h)$, by the Top-down policy via the pointer $PT(N_h)[1]$ we fetch an instance of $\bar{X}$ from the table $TB(A_{h1})$ and bind it to $I_h$. If $I_h$ is null, which means that all answers in the table have already been used by $A_{h1}$ before and the recently derived instance $\bar{I}$ is not new, we backtrack for new answers to $A_{h1}$. Otherwise we continue to prove the resolvent $G_{i+1}$ of $G_i$ and $I_h$.

Case 3 is straightforward, so we move to Case 4. When no table exists for $A_1$, we execute the procedure $create(N_i,A_1)$, which creates a table $(A_1,T,R[M])$ with all $R[j]$ initialized to 1 and $T$ initialized to contain the distinct tuples derived from all facts in $P$ that are instances of $A_1$, and lets $PT(N_i)[1]$ point to the first tuple of $T$ and $PT(N_i)[2]$ to the first rule in $P$ with a head $p(_\ldots)$. When the table $TB(A_1)$ is available, by the Table-first policy and by the Top-down policy, via the pointer $PT(N_i)[1]$ we get an instance $I$ of $\bar{X}$ from $TB(A_1)$. If $I \neq null$, we continue to prove the resolvent $G_{i+1}$ of $G_i$ and $I$. If $I = null$, which shows that all answers in the table have already been used before by the subgoal $A_1$, we try to derive new answers by resolving $A_1$ with rules. There are two distinct cases.

For Case 4.1, $A_1$ has no ancestor subgoal that is its variant, which implies that the derivation does not get into a loop at $N_i$. By the Top-down policy via the pointer $PT(N_i)[2]$ we seek a rule $R_j$ from $P$ whose head is unifiable with $A_1$ and whose status is not “has been used” (i.e. $TB(A_1)[3][j] \neq -1$). If no such a rule exists, $A_1$ is failed and we go backtracking; otherwise we
change the status of $R_j$ into “is being used” (i.e. set $TB(A_1)[j]$ to 0 by executing the procedure $update(N_i,j,0)$) and then continue to prove the resolvant $G_{i+1}$ of $G_i$ and $R_j$.

For Case 4.2, $A_1$ has ancestor subgoals that are its variants, so the derivation has gone into loops. Let $N_h$ be the closest node to $N_i$ whose left-most subgoal $A'_j$ is both a variant and ancestor subgoal of $A_1$. Let the rule being used by $A'_j$ be $R_j$. Then all the rules $R_{ij}$ with $j < i$ whose heads are unifiable with $A_1$ must either have been used before by variant subgoals of $A_1$ (i.e. $TB(A_1)[j][f] = -1$) or be being used by those that are both variants and ancestor subgoals of $A'_j$ (i.e. $TB(A_1)[j][f] = 0$). Clearly, we can not choose $R_j$ (or any $R_{ij}$ with $j < i$ such that $TB(A_1)[j][f] = 0$) to resolve with $A_1$, because that would repeat the loop without producing any new answers to any table subgoals. Therefore, by the Top-down policy via the pointer $PT(N_i)[f]$ we search for a rule $R_k$ next to $R_j$ whose head is unifiable with $A_1$ and whose status is not “has been used”. If $R_k$ exists, we change its status into “is being used” and continue to prove the resolvant $G_{i+1}$ of $G_i$ and $R_k$; otherwise go backtracking.

The Backtracking process (step 3) is as usual, except that the status of the rule $R_j$ needs to be changed after it has been used by the table subgoal $A_f$.

Based on the TP-tree, we have the following standard definitions.

**Definition 3.12** Let $TP_{G_0}$ be a TP-tree of $P \cup \{G_0\}$. All leaves of $TP_{G_0}$ labeled by $\square$ are success leaves and all other leaves are failure leaves. A TP-derivation, denoted by $G_0 \Rightarrow G_1 \Rightarrow \ldots \Rightarrow G_i \Rightarrow \ldots \Rightarrow G_n$, is a partial branch in $TP_{G_0}$ starting at the root, where each $G_i$ is a goal labeling a node $N_i$ and for each $0 \leq i < n$, $G_{i+1}$ is the resolvant of $G_i$ and $G_{i+1}$ with the mgu $\theta_{i+1}$, where $G_{i+1}$ may be a program clause or a table fact or blank (when the left-most subgoal of $G_i$ is a procedure). A TP-derivation is successful if it ends with a success leaf and failed, otherwise. The process of constructing TP-derivations is called TP-resolution.

**Example 3.3** Consider the example program $P_1$ again (see Section 2). By Definitions 3.1 and 3.2, reach is a table predicate and edge is not (see Example 3.1). Now consider applying Algorithm 2 to the top goal $G_0 \leftarrow reach(a,X)$.

By step 1, we set up the root $N_0$. Then by step 2, Case 4 and Case 4.1, we get a table

$$TB(reach(a,X)) : (reach(a,X), \{a\}, \{0,1,1\})$$

and a child node $N_1$ (see Fig.3, where $C_k : D$ represents that the status of the rule $C_k$ is set to $D$). As $reach(a,X)$ is both a variant and ancestor subgoal of $reach(a,Z)$, by step 2 and Case 4.2, the rule $C_2$ is selected, which gives the node $N_2$. Then by Case 2, the answer $reach(a,a)$ is memorized in the table, yielding

$$TB(reach(a,X)) : (reach(a,X), \{(a)\}, \{0,0,1\})$$

and the node $N_3$ is derived using the first table fact. By subsequently performing Cases 3, 2 and 1, we reach a success leaf $N_6$ with the first answer $X = a$ to the top goal. After these steps, the table looks like

$$TB(reach(a,X)) : (reach(a,X), \{(a),(b)\}, \{0,0,1\})$$

Now we go backtracking. By step 3 we will go back until $N_3$. As $C_5$ is not unifiable with the subgoal $edge(a,X)$, by Case 3 and step 3 we come back to $N_1$ from where we consecutively derive a failure leaf $N_7$ (Fig.4), a success leaf $N_{12}$ (Fig.5) and another failure leaf $N_{13}$ (Fig. 6). After those steps, the table becomes

---

*For simplicity, we will not explicitly list the auxiliary subgoal return($\bar{Y}$) unless it is necessarily required.*
TB\((reach(a, X)) : (reach(a, X), \{(a, b, d, e)\}, \{0, -1, -1\})\)

As \(reach(a, Z)\) at \(N_1\) has used all answers in the table and all rules (that is, the pointer \(PT(N_1)[1]\) has reached the end of \(TB\)(reach\((a, X))\)[2] and \(PT(N_1)[2]\) reached the end of the rule base of \(P_1\), we return to the root.

By repeating Case 4 (resolving with a table fact), Case 1 and step 3 twice, we get another two successful derivations as depicted in Fig.7 and Fig.8. Now the table becomes \(TB\)(reach\((a, X)) : (reach(a, X), \{(a, b, d, e)\}, \{-1, -1, -1\})\)

As no more table facts are available to \(N_0\) and all the remaining rules \((C_2\) and \(C_3\)) have already been used by \(N_1\) (i.e. \(TB\)(reach\((a, X)))[3][2] = \(-1\) and \(TB\)(reach\((a, X)))[3][3] = \(-1\)), by Case 4, Case 4.1 and step 3, the answer \(NO\) is returned, which terminates the algorithm. Therefore by putting Figs.1-8 together we obtain the TP-tree \(TP_{G_0}\) of \(P_1 \cup \{G_0\}\).

\[
N_0 := \text{reach}(a, X), \text{return}((X)) \quad /*/\text{The table } TB\text{((reach}(a, X)) \text{is created}
\]

\[
N_1 := \text{reach}(a, Z), \text{edge}(Z, X), \text{memo}(N_0, (X)), \text{lookup}(N_0, (X), I_0), \text{return}((X))
\]

\[
N_2 := \text{memo}(N_1, (a)), \text{lookup}(N_1, (Z), I_1), \text{edge}(Z, X), \text{memo}(N_0, (X)), \text{lookup}(N_0, (X), I_0), \text{return}((X))
\]

- Add \(\text{reach}(a, a)\) to \(TB\)(reach\((a, X))\)
- \(N_1\) gets \(\text{reach}(a, a)\) from \(TB\)(reach\((a, X))\) with mgu \(\{Z/a\}\)

\[
N_3 := \text{edge}(a, X), \text{memo}(N_0, (X)), \text{lookup}(N_0, (X), I_0), \text{return}((X))
\]

\[
N_4 := \text{memo}(N_0, (b)), \text{lookup}(N_0, (X), I_0), \text{return}((X))
\]

- Add \(\text{reach}(a, b)\) to \(TB\)(reach\((a, X))\)
- \(N_0\) gets \(\text{reach}(a, a)\) from \(TB\)(reach\((a, X))\) with mgu \(\{X/a\}\)

\[
N_5 := \text{return}((a))
\]

\[
N_6 := \Box
\]

Fig.3. The first successful TP-derivation with an answer \(X = a\).

\[
N_7 := \text{edge}(b, X), \text{memo}(N_0, (X)), \text{lookup}(N_0, (X), I_0), \text{return}((X))
\]

Fig.4 A failed TP-derivation.
$N_1 : \leftarrow reach(a, Z), edge(Z, X), memo(N_0, (X)), lookup(N_0, (X), I_0), return((X))$

$C_8 : 0$

$N_8 : \leftarrow memo(N_1, (d)), lookup(N_1, (Z, I_1), edge(Z, X), memo(N_0, (X)), lookup(N_0, (X), I_0), return((X))$

Add reach(a, d) to TB[reach(a, X)]

$N_1$ gets reach(a, d) from $TB[reach(a, X)]$ with mgu \{Z/d\}

$N_9 : \leftarrow edge(d, X), memo(N_0, (X)), lookup(N_0, (X), I_0), return((X))$

$C_9$

$N_{10} : \leftarrow memo(N_0, (e)), lookup(N_0, (X), I_0), return((X))$

Add reach(a, e) to $TB[reach(a, X)]$

$N_0$ gets reach(a, b) from $TB[reach(a, X)]$ with mgu \{X/b\}

$N_{11} : \leftarrow return((b))$

Return $X = b$

$N_{12} : \Box$

Fig.5 The second successful TP-derivation with the second answer $X = b$.

$N_1 : \leftarrow reach(a, Z), edge(Z, X), memo(N_0, (X)), lookup(N_0, (X), I_0), return((X))$

$C_3 : -1$

$N_1$ gets reach(a, e) from $TB[reach(a, X)]$ with mgu \{Z/e\}

$N_{13} : \leftarrow edge(e, X), memo(N_0, (X)), lookup(N_0, (X), I_0), return((X))$

Fig.6 Another failed TP-derivation.

$N_0 : \leftarrow reach(a, X), return((X))$

$C_1 : -1$

$N_0$ gets reach(a, d) from $TB[reach(a, X)]$ with mgu \{X/d\}

$N_{14} : \leftarrow return((d))$

Return $X = d$

$N_{15} : \Box$

Fig.7 The third successful TP-derivation with the third answer $X = d$.

$N_0 : \leftarrow reach(a, X), return((X))$

$N_0$ gets reach(a, e) from $TB[reach(a, X)]$ with mgu \{X/e\}

$N_{16} : \leftarrow return((e))$

Return $X = e$

$N_{17} : \Box$

Fig.8 The fourth successful TP-derivation with the fourth answer $X = e$.

4 Characteristics of TP-Resolution

In this section, we prove the termination of Algorithm 2 and the soundness and completeness of TP-resolution. We also discuss the way to deal with the cut operator in TP-resolution.
4.1 Soundness and Completeness

In order to guarantee termination of Algorithm 2, we restrict ourselves to logic programs with the bounded-term-size property. Informally, (in our context) a logic program \( P \) has the bounded-term-size property if for any top goal \( G_0 \), the TP-tree \( TP_{G_0} \) of \( P \cup \{ G_0 \} \) contains no subgoals with infinitely large size terms. Obviously, all function-free logic programs have the bounded-term-size property.

**Theorem 4.1 (Termination)** Let \( P \) be a logic program with the bounded-term-size property and \( G_0 \) a top goal. Algorithm 2 terminates with a finite TP-tree.

The following lemma is required to prove this theorem.

**Lemma 4.2** Let \( G_i \) and \( G_k \) be two goals in a TP-derivation of \( P \cup \{ G_0 \} \) and \( A \) and \( A' \) be the left-most subgoals of \( G_i \) and \( G_k \), respectively. If \( A \) is both an ancestor subgoal and a variant of \( A' \), then \( A \) is a table subgoal.

**Proof.** Let \( A \) have the predicate \( p \). By Definitions 3.9 and 3.10, \( A \) being both an ancestor subgoal and a variant of \( A' \) implies that there is a call path in \( P \) starting from \( B \) and ending at \( B' \), where \( B \) is the head of a clause in \( P \) with a non-empty body and a variant of \( B' \), and \( A \) and \( A' \) are instances of \( B \). So there must be a cycle in the atoms-call graph \( AG_P \) that contains a node \( B \) or its variant. Then by Definition 3.2 \( p \) is a table predicate. Hence, \( A \) is a table subgoal. □

**Proof of Theorem 4.1.** Assume, on the contrary, that Algorithm 2 does not terminate. Then it generates an infinite TP-tree. This can occur only in the two cases: it memorizes infinitely many (new) answers in tables (so it will do backtracking infinite times) or traps into an infinite derivation. We first show that the first case is not possible. Since \( P \) has the bounded-term-size property, all table facts have finite size. As \( P \) contains a finite set of predicate symbols, function symbols and constants, any table fact having finite size implies that any table contains a finite set of table facts.

We now assume the second case. For the same reason as the first case, since \( P \) has the bounded-term-size property and contains a finite set of clauses, any infinite derivation must contain an infinite loop, i.e. an infinite set of subgoals, \( A_0, A_1, \ldots, A_k, \ldots \), such that for any \( i \geq 0 \), \( A_i \) is both an ancestor subgoal and a variant of \( A_{i+1} \). This means that all the \( A_i \)'s are table subgoals (Lemma 4.2). However, from Cases 4 and 4.2 we see that such a set of subgoals will never be generated by Algorithm 2 unless \( P \) contains an infinite set of rules whose heads are unifiable with the \( A_i \)'s, a contradiction. □

**Theorem 4.3 (Soundness and Completeness)** Let \( P \) be a logic program with the bounded-term-size property, \( G_0 \) a top goal and \( TP_{G_0} \) the TP-tree of \( P \cup \{ G_0 + \text{return}(\overline{Y}) \} \). Let \( ST_{G_0} \) be the SLD-tree of \( P \cup \{ G_0 + \text{return}(\overline{Y}) \} \) via the left-most computation rule. \( TP_{G_0} \) and \( ST_{G_0} \) have the same set of answers to \( G_0 \).

**Proof.** The soundness is obvious because each successful TP-derivation is equivalent to a certain successful SLD-derivation under the PMF mode.

To show the completeness of Algorithm 2, first note that when \( ST_{G_0} \) contains no infinite loops, Algorithm 1 traverses all its branches and terminates. Algorithm 2 is Algorithm 1 with the following enhancements.
1. All answers to table subgoals are sequentially tabled and consumed (in the same order). Let $A_1$ be a table subgoal at a node $N_i$ labeled with a goal $G_i = \leftarrow A_1, ..., A_m$. Then all the facts in $P$ that are instances of $A_1$ are tabled in $TB(A_1)$ (see Case 4 in Algorithm 2 and create( ) in Definition 3.5). Moreover, for each rule $R_j = A'_1 \leftarrow B_1, ..., B_n$ in $P$ with $A_1 \theta = A'_1 \theta$, the resolvent of $G_i$ and $R_j$ is $\leftarrow (B_1, ..., B_n) \theta$, $memo(N_i, \bar{X} \theta)$, $lookup(N_i, \bar{X}, I)$, $A_2, ..., A_m$ (see Cases 4.1 and 4.2 and Definition 3.6), so that (by the PMF mode) all answers to $A_1$ that result from the proof of $(B_1, ..., B_n) \theta$ are added to $TB(A_1)$ (see Case 2 and Definition 3.6). All answers to $A_1$ in $TB(A_1)$ are applied to the remaining subgoals of $G_i$ in the same order as they were generated (see Cases 4 and 2). Obviously, such an enhancement preserves the original answer set to $G_0$.

2. Reappplication of rules to variant table subgoals is excluded by keeping the status of the rules. Observe that for any table subgoal $A_1$, its proof is reduced to the proof of $(B_1, ..., B_n) \theta$, $memo(\bar{X} \theta)$, $lookup(\bar{X}, I)$ when we apply to $A_1$ a rule $R_j$ of the form $A'_1 \leftarrow B_1, ..., B_n$ with $A_1 \theta = A'_1 \theta$ (see Definition 3.6). So each time $(B_1, ..., B_n) \theta$ is proved true with an mgu $\theta_1$, an answer $A_1 \theta_1$ will be added by $memo(\bar{X} \theta \theta_1)$ to the table $TB(A_1)$ if it is new. In consequence, when we leave $R_j$ for alternative rules for $A_1$, which means that the answers to $(B_1, ..., B_n) \theta$ have been exhausted via backtracking, $TB(A_1)$ contains all answers to $A_1$ derived by applying $R_j$. Therefore, when a variant subgoal $A$ of $A_1$ occurs later, $A$ can readly fetch the answers from $TB(A_1)$, rather than to recompute them through $R_j$. Algorithm 2 achieves this effect by keeping the status of $R_j$ (and all rules related to table subgoals). The initial status of $R_j$ is 1, showing that it has never been tried by any subgoals (see Case 4 and Definition 3.5 for create( )). When $R_j$ is applied to $A_1$, its status is changed to 0, showing that it is being used (see Cases 4.1 and 4.2). When $A_1$ leaves $R_j$ for alternative rules (via backtracking), the status of $R_j$ is changed to -1, showing that it has been used (see step 3). As a result, after $R_j$ has been used by $A_1$, it will never be re-applied to any variant subgoal $A$ of $A_1$ (see Cases 4.1 and 4.2). Instead, $A$ directly gets those answers from $TB(A_1)$ (see Case 4). Clearly, such an enhancement preserves the original answer set to $G_0$.

3. All infinite loops are broken by skipping the rules that are being used by variant ancestor subgoals (i.e. ancestor goals that are variants). Suppose that Algorithm 2 has generated a TP-derivation as depicted in Fig.9, where $A_1$ is the closest variant ancestor subgoal of $A'_1$. We see that a loop occurs. By Lemma 4.2, $A_1$ and $A'_1$ are table subgoals. Now consider expanding $A'_1$. By the Top-down policy (see TP-strategy), all rules $R_f$ with $0 \leq f < j$ that are unifiable with $A'_1$ must have been used before by some variant subgoals of $A'_1$. So instead of reapplying these rules, $A'_1$ directly uses the answers in $TB(A_1)$ (see Case 4).

Assume that all answers in $TB(A_1)$ have been used by $A'_1$ (via backtracking). By Case 4.2, instead of selecting $R_j$, $A'_1$ jumps to using $R_k$ ($k > j$) which is the choice rule of $A_1$ (i.e. $A_1$ will use $R_k$ after leaving $R_j$). We justify such a decision as follows. Assume, on the contrary, that we continue choosing $R_j$ to expand $A'_1$. As $A'_1$ and $A_1$ are variant subgoals, the TP-tree $T_{A'_1}$ starting from $N_k$ via $R_j$ will grow in the same way as the TP-tree $T_{A_1}$ starting from $N_i$ via $R_j$ except that some branches in $T_{A_1}$ that are generated by applying some rules will be cut off in $T_{A'_1}$ because all answers derived from these rules have been recorded in the tables. So after some steps the TP-derivation in Fig.9 will extend to the TP-derivation in Fig.10, where $A'_1$ is the closest variant ancestor subgoal of $A''_1$. Let $T_{A_1}$ consist of all TP-derivations before applying $R_j$ to $A'_1$ and $T_{A'_1}$ consist of all TP-derivations before applying $R_j$ to $A''_1$ (Fig.11). Then the derivations made in $T_{A'_1}$ must be a subset of those made in $T_{A_1}$. As all answers to the table subgoals obtained in $T_{A'_1}$ have been tabled and consumed in $T_{A_1}$, which means that no new answers can be derived by applying these answers to the rules used in $T_{A_1}$, the
derivations in $T_{A'_1}$ will produce no new answers to any subgoals. Therefore, $T_{A'_1}$ is redundant and should be pruned, which justifies our decision to skip $R_j$. That is, by skipping the rules that are being used by variant ancestor subgoals, we break all infinite loops without losing any answers to $G_0$.

To sum up, Algorithm 2 enhances Algorithm 1 by tabling all answers to table subgoals, by keeping the status of all rules related to table subgoals to avoid reapplying them to variant table subgoals, and by skipping the rules that are being used by variant ancestor subgoals to break infinite loops, while preserving the original answers to the top goal $G_0$. So when Algorithm 2 terminates, it traverses all but redundant branches of $ST_{G_0}$. Hence, $TP_{G_0}$ and $ST_{G_0}$ have the same set of answers to $G_0$. □

\[
\begin{aligned}
N_i &:= A_1, A_2 - A_m \\
&\downarrow R_j : 0 \\
N_k &:= A'_1, B_1 - B_n, A_2 - A_m \\
&\downarrow R_j \\
N_i &:= A''_1, D_1 - D_1, B_1 - B_n, A_2 - A_m \\
&\downarrow R_j \\

T_{A_1} &:= A_1, A_2 - A_m \\
&\downarrow R_j : 0 \\
N_i &:= A'_1, B_1 - B_n, A_2 - A_m \\
&\downarrow R_j \\
N_i &:= A''_1, D_1 - D_1, B_1 - B_n, A_2 - A_m \\
&\downarrow R_j \\

\end{aligned}
\]

Fig. 9 \hspace{1cm} Fig. 10 \hspace{1cm} Fig. 11

4.2 Dealing with Cuts

The cut operator, !, is very popular in Prolog programs. It basically serves two purposes. One is to simulate the if-then-else statement, which is one of the key control flow statements in procedural languages. For example, in order to realize the statement if $A$ then $B$ else $C$, we define the following predicate:

\[
p \leftarrow A, !, B. \\
p \leftarrow C.
\]

The other purpose the cut operator serves is to prune the search space by aborting further exploration of some remaining branches, which may lead to significant computation savings. For instance, the following clauses

\[
p(X) \leftarrow A_1, \ldots, A_m, !.
\]

Remaining rules defining $p(X)$.
achieve the effect that for any $X$ whenever $A_1, \ldots, A_m$ is true with an mgu $\theta$, we return $p(X)\theta$ and stop searching the remaining space (via backtracking on the $A_i$s and using the remaining rules of $p(X)$) for any additional answers to $p(X)$.

The cut operator requires a strictly sequential strategy — Prolog-strategy for the selection of goals, subgoals and program clauses. TP-strategy is an enhancement of Prolog-strategy with the two policies for dealing with table facts: Table-first when both table facts and program clauses are available and Top-down for the selection of table facts. As new answers are always appended to the end of a table, by the PMF mode such an enhancement does not affect the original sequentiality of Prolog-strategy. That is, TP-strategy supports the cut operator as well. We now discuss how to realize cuts in TP-resolution.

Let $G_i = \leftarrow A_1, \ldots, A_m$ be a goal at a node $N_i$ and $R_j = A'_1 \leftarrow B_1, \ldots, B_k, !, \ldots$ a rule with $A_1\theta = A'_1\theta$. Then the resolvant of $G_i$ and $R_j$ is $G_{i+1} = \leftarrow (B_1, \ldots, B_k)\theta, !, \ldots$ When evaluated as a subgoal for forward node expansion, $!$ is unconditionally true. However, during backtracking, it will skip all $B_k$s and jump back to the parent node of $N_i$. Since $R_j$ has been used by $A_1$ after the backtracking and the remaining rules related to $A_1$ will never be allowed to use by $A_1$ or its variant subgoals, the status of all $R_k$s with $l \geq j$ in $TB(A_1)$ should be set to $-1$. In order to formulate such actions, we attach the node name $N_i$ to $!$ (as a directive for backtracking). That is, we have a subgoal $!(N_i)$, instead of $!$, in the resolvant $G_{i+1}$. Then the cut operator can be realized in TP-resolution by the following slight extension to Algorithm 2\(^9\).

1. Add before Case 1 the following:
   - Case 0: $A_1$ is $(N_h)$. Set $G_{i+1}$ to $\leftarrow A_2, \ldots, A_m$ and goto 2.

2. In 3 (Backtracking), add before “If $A_{f_1}$ is $\text{memo}(.)$” the statement: If $A_{f_1}$ is $(N_h)$, let the left-most subgoal at $N_h$ be $A_{h_1}$ and the rule used by $A_{h_1}$ to produce the subgoal $!(N_h)$ be $R_j$, set $TB(A_{h_1})[3][4]$ to $-1$ for all $l \geq j$ (if $A_{h_1}$ is a table subgoal) and goto 3 with $N = N_h$.

It is easy to verify that (the extended) Algorithm 2 achieves the same effects of cuts as Prolog (Algorithm 1 with a similar extension) except for the situations in which Prolog goes into an infinite loop. To illustrate this, consider the following example.

**Example 4.1** The following two clauses

\[
\begin{align*}
\text{not}_{\mathcal{P}}(X) & \leftarrow p(X), !, \text{fail}. & \text{C}_1 \\
\text{not}_{\mathcal{P}}(X). & \text{C}_2
\end{align*}
\]

define the predicate $\text{not}_{\mathcal{P}}$ which says that for any object $X$, $\text{not}_{\mathcal{P}}(X)$ succeeds if and only if $p(X)$ fails. Let $G_0 = \leftarrow \text{not}_{\mathcal{P}}(a)$ be the top goal and the programs $P_i$ be defined as follows.

1. $P_1 = \{C_1, C_2\}$. As $p(a)$ fails, $C_2$ is applied, so that both Prolog and Algorithm 2 give an answer $\text{YES}$ to $G_0$.

2. $P_2 = \{C_1, C_2, p(a)\}$. As $p(a)$ succeeds, the cut operator $!$ in $C_1$ is executed. Since the subgoal $\text{fail}$ always fails, the backtracking on $!$ disenables $C_2$, so that both Prolog and Algorithm 2 give an answer $\text{NO}$ to $G_0$.

\(^9\)Similar extension can be made to Algorithm 1.
3. \( P_3 = \{C_1, C_2, p(X) \leftarrow p(X)\} \). As Prolog goes into an infinite loop in proving the subgoal \( p(a) \), no answer can be obtained. However, Algorithm 2 breaks the loop by making \( p(a) \) fail, so that \( C_2 \) is applied, which leads to an answer YES to \( C_0 \).

It should be pointed out that as Algorithm 2 breaks infinite loops by trying alternative rules, we must be careful in using cuts to simulate the if-then-else function. To see this, consider the program \( P \) given by

\[
A \leftarrow A, !, B.
A \leftarrow C.
C.
\]

We can not use this program to represent if \( A \) then \( B \) else \( C \) because interpreting it by Prolog will go into an infinite loop \( A \leftarrow A, !, B \leftarrow A, ... \), whereas running it by Algorithm 2 will lead to a derivation like \( A \leftarrow A \Rightarrow A, !((N_0), B \leftarrow C, !((N_0), B \Rightarrow !((N_0), B \Rightarrow B \) in which both \( C \) and \( B \) are executed, which violates the condition that they are exclusive events.

5 Conclusions

Existing tabulated resolutions, such as OLDT-resolution, SLG-resolution and Tabulated SLS-resolution, rely on the solution-lookup mode in formulating tabling. As lookup nodes are not allowed to resolve table subgoals against program clauses, the underlying tabulated resolutions can not be linear, so that it is impossible to implement such resolutions using a simple stack-based memory structure like that in Prolog. This may make their implementation much more complicated (XSB is a typical example [12], in contrast to Prolog [20, 22]). Moreover, As lookup nodes fully depend on solution nodes, without any autonomy, it is difficult to guarantee that some strictly sequential operators in Prolog such as cuts work normally ([11, 13]).

In contrast, TP-resolution presented in this paper has the following novel properties.

1. It does not distinguish between solution and lookup nodes. Any nodes can resolve table subgoals against program clauses as well as answers in tables provided that they abide by the Table-first policy, regardless of when and where they are generated.

2. It makes linear tabulated derivations based on TP-strategy in the same way as Prolog except that infinite loops are broken and redundant computations are avoided. The resolution algorithm (Algorithm 2) is simple and can be implemented by a slight extension to any existing Prolog abstract machines such as WAM [20] or ATOAM [22].

3. It deals with cuts as effectively as Prolog.

We are now working on two further tasks. One is to extend TP-resolution to compute the well-founded semantics of general logic programs, and the other is to implement TP-resolution to obtain a tabulated Prolog system.

Acknowledgements: The research is supported in part by Chinese National Natural Science Foundation and Chinese State Education Commission Foundation.
References


