Linear Tabulated Resolution for the Well-Founded Semantics

Yi-Dong Shen*[1], Li-Yan Yuan[2], Jia-Huai You[2], and Neng-Fa Zhou[3]

[1] Department of Computer Science, Chongqing University,
Chongqing 400044, P.R.China,
ydshen@cqu.edu.cn
[2] Department of Computing Science, University of Alberta,
Edmonton, Alberta, Canada T6G 2H1,
{yuan, you}@cs.ualberta.ca
[3] Department of Computer and Information Science, Brooklyn College,
The City University of New York, New York, NY 11210-2889, USA,
zhou@sci.brooklyn.cuny.edu

Abstract. Global SLS-resolution and SLG-resolution are two representative mechanisms for top-down evaluation of the well-founded semantics of general logic programs. Global SLS-resolution is linear but suffers from infinite loops and redundant computations. In contrast, SLG-resolution resolves infinite loops and redundant computations by means of tabling, but it is not linear. The distinctive advantage of a linear approach is that it can be implemented using a simple, efficient stack-based memory structure like that in Prolog. In this paper we present a linear tabulated resolution for the well-founded semantics, which resolves the problems of infinite loops and redundant computations while preserving the linearity. For non-floundering queries, the proposed method is sound and complete for general logic programs with the bounded-term-size property.

1 Introduction

Two representative methods have been presented in literature for top-down evaluation of the well-founded semantics of general logic programs: Global SLS-resolution [5,6] and SLG-resolution [2,3]. Global SLS-resolution is a direct extension to SLDNF-resolution [4], which treats infinite derivations as *failed* and infinite recursions through negation as *undefined*. Like SLDNF-resolution, it is *linear* in the sense that for any derivation $G_0 \Rightarrow_{C_1,\theta_1} G_1 \Rightarrow ... \Rightarrow_{C_i,\theta_i} G_i$ with G_i the latest generated goal, it makes the next derivation step either by expanding G_i by resolving a subgoal in G_i with a program clause, i.e. $G_i \Rightarrow_{C_{i+1},\theta_{i+1}} G_{i+1}$, or by expanding G_{i-1} via backtracking. The distinctive advantage of a linear approach is that it can be implemented using a simple, efficient stack-based memory structure (like that in Prolog). However, Global SLS-resolution inherits

* Currently on leave at Department of Computing Science, University of Alberta, Canada. Email: ydshen@cs.ualberta.ca

M. Gelfond, N. Leone, G. Pfeifer (Eds.): LPNMR'99, LNAI 1730, pp. 192–205, 1999.

from SLDNF-resolution the two serious problems: infinite loops and redundant computations.

SLG-resolution (similarly, Tabulated SLS-resolution [1]) is a tabling mechanism for top-down evaluation of the well-founded semantics. The main idea of tabling is to store intermediate results of relevant subgoals and then use them to solve variants of the subgoals whenever needed. Since no variant subgoals will be recomputed by applying the same set of program clauses, infinite loops can be avoided and redundant computations be substantially reduced. Like other existing tabling mechanisms, SLG-resolution adopts the *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 the tables. As an illustration, consider the derivation $p(X) \Rightarrow_{C_{p_1},\theta_1} q(X) \Rightarrow_{C_{q_1},\theta_2} p(Y)$. Assume that no answers of $p(X)$ have been derived. Since $p(Y)$ is a variant of $p(X)$ and thus a lookup node, the next derivation step is to expand $p(X)$ against a program clause, instead of expanding the latest generated goal $p(Y)$. Apparently, such a derivation is not linear. Because of such non-linearity, SLG-resolution can neither be implemented using an efficient stack-based memory structure nor utilize those useful strictly sequential operators such as cuts in Prolog. This has been evidenced by the fact that a well-known tabling system, XSB, which is an implementation of SLG-resolution [7,8,9], disallows clauses like $p(.) \leftarrow ..., t(.), !, ...$ where $t(.)$ is a tabled subgoal, because the tabled predicate t occurs in the scope of a cut [9].

The objective of our research is to develop a linear tabling method for top-down evaluation of the well-founded semantics of general logic programs, which resolves infinite loops and redundant computations, without sacrificing the linearity of SLDNF-resolution. In an earlier paper [11], we presented a linear tabling mechanism called *TP-resolution* for positive logic programs ("TP" for "Tabulated Prolog"). In TP-resolution, each node in a search tree can act both as a solution node and as a lookup node, regardless of when and where it is generated. This represents an essential difference from existing tabling approaches. The main idea is as follows: For any selected subgoal A at a node N_i labeled with a goal G_i, we first try to use an answer I in the table of A to generate a child node N_{i+1}, which is labeled by the resolvant of G_i and I. If such answers are not available in the table, we then resolve A against program clauses in a top-down order, except for the case where the derivation has stepped into a loop at N_i. In such a case, the subgoal A will skip the clause that is being used by its ancestor subgoal that is a variant of A. For example, for the derivation $p(X) \Rightarrow_{C_{p_1},\theta_1} q(X) \Rightarrow_{C_{q_1},\theta_2} p(Y)$, we will expand $p(Y)$ by resolving it against the program clause next to C_{p_1}. Thanks to its linearity, TP-resolution can be implemented by an extension to any existing Prolog abstract machines such as WAM [14] or ATOAM [15].

In this paper, we extend TP-resolution to *TPWF-resolution*, which computes the well-founded semantics of general logic programs. The extension is non-trivial because of possible infinite recursions through negation. In addition to

the strategy for clause selection adopted by TP-resolution, TPWF-resolution uses two critical mechanisms to deal with infinite recursions through negation. One is *making assumptions* for negative loop subgoals whose truth values are currently undecided, and the other is *doing answer iteration* to derive complete answers of loop subgoals. For non-floundered queries, TPWF-resolution is sound and complete for general logic programs with the bounded-term-size property.

Section 2 will give an illustrative example to outline these main ideas, and Section 3 defines TPWF-trees based on these strategies. Section 4 presents the definition of TPWF-resolution and discusses its properties.

1.1 Notation and Terminology

Variables begin with a capital letter, and predicates, functions and constants with a lower case letter. By $\boldsymbol{E}$ we denote a list/tuple $(E_1, ..., E_m)$ of elements. Let $\boldsymbol{X} = (X_1, ..., X_m)$ be a tuple of variables and $\boldsymbol{I} = (I_1, ..., I_m)$ a tuple of terms. By $\boldsymbol{X}/\boldsymbol{I}$ we denote an mgu $\{X_1/I_1, ..., X_m/I_m\}$. By $p(.)$ we refer to any atom with the predicate p and $p(\boldsymbol{X})$ to an atom $p(.)$ that contains the list $\boldsymbol{X}$ of distinct variables. For instance, if $p(\boldsymbol{X}) = p(W, a, f(Y, W), Z)$, then $\boldsymbol{X} = (W, Y, Z)$.

By a *variant* of an atom (resp. subgoal or term) A we mean an atom (resp. subgoal or term) A' that is the same as A up to variable renaming.[1] A set of atoms (resp. subgoals or terms) that are variants of each other are called *variant atoms* (resp. *variant subgoals* or *variant terms*). Moreover, for any element E by E being in a set S we understand a variant of E is in S.

For convenience of describing our method, we use the four truth values: t (true), f (false), u (undefined), and u^* (temporarily undefined), with $\neg t = f$, $\neg f = t$, $\neg u = u$, and $\neg u^* = u^*$. As its name suggests, u^* will be used as a temporary truth value when the truth value (t, f or u) of a subgoal is currently undecided (due to the occurrence of loops). In addition to $f \wedge V = f$ and $t \wedge V = V$ for any $V \in \{t, f, u, u^*\}$, we have $u \wedge u^* = u^*$. Let A be an atom. By A^* we refer to an answer A with truth value u^*.

Finally, clauses in a program with the same head predicate p are numbered sequentially, with C_{p_i} referring to its i-th clause ($i > 0$).

2 Main Ideas

In this section, we outline the main ideas of TPWF-resolution through an illustrative example.

Example 1. Consider the following program:

P_1: $p(X) \leftarrow q(X)$. C_{p_1}
$p(a)$. C_{p_2}
$q(X) \leftarrow \neg r$. C_{q_1}
$q(X) \leftarrow w$. C_{q_2}

[1] By this definition, A is a variant of itself.

$q(X) \leftarrow p(X).$ C_{q_3}
$r \leftarrow \neg s.$ C_{r_1}
$s \leftarrow \neg r.$ C_{s_1}
$w \leftarrow \neg w, v.$ C_{w_1}

Let $G_0 = \leftarrow p(X)$ be the query (top goal). Reasoning in the same way as Prolog,[2] we successively generate the nodes $N_0 - N_7$ as shown in Fig. 1. Obviously Prolog will repeat the loop between N_3 and N_7 infinitely. However, we break the loop by *disallowing* N_7 to select the clause C_{r_1}, which is being used by N_3. This makes N_7 have no clause to unify with, which leads to backtracking. Since the loop is negative in the sense that it goes through negation, N_7 should not be failed by falsifying r at this moment. Instead, r is *assumed* to be temporarily undefined (i.e. $r = u^*$). By definition $r = u^*$ (at N_7) means $\neg r = u^*$ (at N_6), so that $s = u^*$ (at N_5) is *derived*. For the same reason, $r = u^*$ (at N_3) is derived.

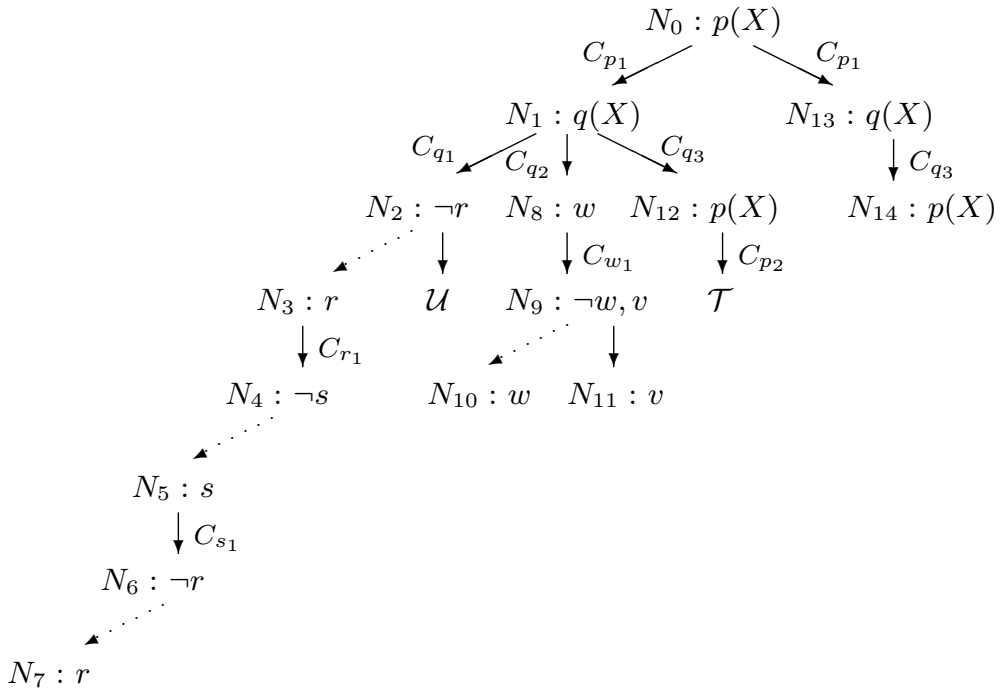

Fig. 1. TPWF-derivations.

We use two data structures, UA and UD, to keep atoms that are assumed and derived to be temporarily undefined, respectively. Therefore, after these steps, $UA = \{r\}$ and $UD = \{s, r\}$. We are then back to N_3.

Since N_3 is the top node of the loop, before failing it via backtracking we need to be sure that r has got its complete set of answers ($r = t$ or $r = u$ or $r = f$). This is achieved by performing *answer iteration* via the loop. That is, we regenerate the loop to see if any new answers can be derived until we reach

[2] That is, we use the following control strategy: **Depth-first** (for goal selection) + **Left-most** (for subgoal selection) + **Top-down** (for clause selection) + **Last-first** (for backtracking).

a fixpoint. We use a flag variable NEW, with $NEW = 0$ initially. Whenever a new answer with truth value t or u for any subgoal is derived, NEW is set to 1. Before starting an iterate, we set $NEW = 0$ and $UA = UD = \{\}$. The answer iteration stops by the end of some iterate, where $NEW = 0$ and ($UA \subseteq UD$ or $UD = \{\}$). The fact that $NEW = 0$ and $UA \subseteq UD$ indicates that *the truth values of all atoms in UA totally depend on how they are assumed in the negative loop*, which, under the well-founded semantics [12], amounts to saying that these truth values are undefined.

Since up to now no answer with truth value t or u has been derived (i.e. $NEW = 0$) and $UA = \{r\} \subset UD = \{s, r\}$, the termination condition of answer iteration is satisfied. Therefore, we change the truth values of all atoms in UD from temporarily undefined to undefined (i.e. $r = s = u$) and memorize the new answers in respective tables. After the completion of answer iteration, we set $UA = UD = \{\}$.

By definition, $r = u$ (at N_3) means $\neg r = u$ (at N_2), which leads to an answer node $\mathcal{U}$ for the top goal (see Fig. 1). That is, we have $q(X) = u$ and $p(X) = u$, which are memorized in their tables.

Now we backtrack $q(X)$ at N_1. Applying C_{q_2} and C_{w_1} leads to $N_8 - N_{10}$, which forms another negative loop. In the same way as above, we assume $w = u^*$ and put w into UA. So $\neg w = u^*$, which leads to the node N_{11}. Since v is false, we backtrack to N_9 and then to N_8, with $NEW = 0$, $UA = \{w\}$ and $UD = \{\}$. Again, before leaving N_8 via backtracking, we need to complete the answers of w by means of answer iteration via the loop. Obviously, the termination condition of answer iteration is satisfied. Here $NEW = 0$, $w \in UA$ and $w \notin UD$ suggests that *w can not be inferred from the program whatever truth values we assign to the temporarily undecided subgoals in UA*. This, under the well-founded semantics, implies that w is false. So we set $w = f$ and come back to N_1 again.

Applying C_{q_3} leads to N_{12}. We see that there is a loop between N_0 and N_{12}. Instead of selecting C_{p_1} which is being used by N_0, we use C_{p_2} to unify against $p(X)$, which leads to an answer node $\mathcal{T}$ with mgu X/a. That is, $p(a) = t$ and $q(a) = t$, which are added to the tables of $p(X)$ and $q(X)$, respectively (NEW is then set to 1).

Since the loop $N_0 \rightarrow N_1 \rightarrow N_{12}$ is positive, we backtrack to N_1 and then to N_0, *making no assumption*. This time, we have $NEW = 1$, $UA = \{\}$ and $UD = \{\}$. Since N_0 is the top loop node and $NEW = 1$, we do answer iteration by regenerating the loop, which leads to $N_0 \rightarrow N_{13} \rightarrow N_{14}$. Since C_{p_1} is being used by N_0 and C_{p_2} has already been used before (by N_{12}, with the answer stored in the table of $p(X)$), $p(X)$ at N_{14} has no clause to unify with. So we backtrack to N_{13} and then to N_0. Now, $NEW = 0$ and $UA = UD = \{\}$, so we end the iteration.

Since N_0 is the root, the evaluation of G_0 terminates. The derived answers are: $p(a) = q(a) = t$, $p(b) = q(b) = u$ for any $b \neq a$, $r = s = u$, and $w = v = f$. We see that these answers constitute the well-founded model for P_1. □

The tabulated resolution shown in Example 1 is obviously linear. Meanwhile, we see that it resolves infinite loops and redundant computations without losing any answers. Main points are summarized as follows:

1. Tabling. Tables are used to store intermediate results, which is the basis of all tabulated resolutions.
2. Clause selection. Without loops, clauses are selected in the same way as in Prolog except that clauses that have been used before will not be reapplied because the complete set of answers derived via those clauses has already been memorized in related tables. For example, N_{14} skips C_{p_2} because the clause has already been used by N_{12}. This avoids redundant computations. When a loop occurs, however, clauses that are being used by ancestor loop subgoals will be skipped. For example, C_{r_1}, C_{w_1} and C_{p_1} are skipped by N_7, N_{10} and N_{12}, respectively. This breaks infinite loops.
3. Assumption. For a positive loop subgoal, backtracking proceeds in the same way as in Prolog (see N_{12} and N_{14}). A negative loop subgoal whose truth value is currently undecided, however, will be assumed temporarily undefined before being failed (see N_7 and N_{10}). Temporarily undefined values will be removed (from tables) when their t or u counterparts are derived. This guarantees the correctness of answers.
4. Answer iteration. Before leaving a loop by failing its top loop node (e.g. N_3, N_8 and N_0), iteration will be carried out to derive complete answers of loop subgoals. Without iteration, we would miss answers because some clauses have been skipped to break infinite loops. The process of answer iteration is briefly described as follows. Let N_t be the top loop node. We first check if the termination condition is satisfied (i.e. $NEW = 0$ and ($UA \subseteq UD$ or $UD = \{\}$)). If not, we start an iterate by setting $NEW = 0$ and $UA = UD = \{\}$. The iterate will regenerate the loop (e.g. $N_0 \rightarrow N_{13} \rightarrow N_{14}$ in Fig. 1). During the iterate, NEW, UA and UD will be updated accordingly. By the end of the iterate, i.e. when we come back to the top loop node N_t again and try to fail it via backtracking, we distinguish among the following cases:
   - $NEW = 1$, which means at least one new answer, with truth value t or u, has been derived (and added to the related table) during the iterate. So we start a new iterate to seek more answers.
   - $NEW = 0$ and ($UA \subseteq UD$ or $UD = \{\}$). Stop the iteration with all temporarily undefined answers replaced by undefined ones. After this, the answers of all subgoals involved in the loop are completed (i.e. the tables of these subgoals contain all of their answers). We attach a flag $COMP$ to each table with $COMP = 1$ standing for being completed. For any subgoal A whose table flag $COMP$ is 1, its instance A' is true if $A' = t$ is in the table of A, undefined if $A' = u$ is in the table but $A' = t$ is not, and false if neither is in the table. For instance, in the above example, $p(a) = t$ and $p(X) = u$ being in the table of $p(X)$ shows that $p(a)$ is true and $p(b)$ is undefined for any $b \neq a$.
   - Otherwise. Let $UC = UA - UD$. Since $NEW = 0$, for all subgoals in UC we can not infer any new answers for them from the program whatever

truth valus we assign to the temporarily undecided subgoals in UA. This implies that the answers of these subgoals have been completed, so we set the flag $COMP$ of their tables to 1. Since the subgoals in $UD - UC$ are still temporarilly undecided, we start next iterate. The iteration will terminate provided that the program has the bounded-term-size property [13].

3 TPWF-Trees

In this section we define TPWF-trees, which is the basis of TPWF-resolution. We begin by defining tables.

3.1 Tables

Let P be a logic program and $p(\boldsymbol{X})$ an atom. Let P contain exactly N_p clauses with a head $p(.)$. A *table* for $p(\boldsymbol{X})$, denoted $TB(p(\boldsymbol{X}))$, is a four-tuple $(p(\boldsymbol{X}), T, C, COMP)$, where

1. $T = \{T_1, T_2\}$, with T_1 and T_2 storing answers of $p(\boldsymbol{X})$ with truth values t and u, respectively.
2. C is a vector of N_p elements, keeping the status of C_{p_i}s w.r.t. $p(\boldsymbol{X})$. $C[i] = 0$ (resp. $= 1$) represents that the clause C_{p_i} is *no longer available* (resp. *still available*) to $p(\boldsymbol{X})$.
3. $COMP \in \{0, 1\}$, with $COMP = 1$ indicating that the answers of $p(\boldsymbol{X})$ have been completed.

For convenience, we use $TB(p(\boldsymbol{X})) \rightarrow t\_answer[i]$ and $TB(p(\boldsymbol{X})) \rightarrow u\_answer[i]$ to refer to the i-th answer in T_1 and T_2, respectively, $TB(p(\boldsymbol{X})) \rightarrow clause\_status[i]$ to refer to the status of C_{p_i} w.r.t. $p(\boldsymbol{X})$, and $TB(p(\boldsymbol{X})) \rightarrow COMP$ to refer to the flag $COMP$.

When a table $TB(p(\boldsymbol{X}))$ is created, $T_1 = T_2 = \{\}$, the status of all clauses is initialized to 1, and $COMP = 0$. Answers in a table will be read sequentially from T_1 followed by T_2. When $T_1 = T_2 = \{\}$ and $COMP = 1$, $p(\boldsymbol{X}) = f$.

Example 2. Consider again the program P_1 in Example 1. After node N_{14} is generated (see Fig. 1), we have the following tables:

$TB(p(X))$: $(p(X), \{\{p(a)\}, \{p(X)\}\}, \{1, 0\}, 0)$,
$TB(q(X))$: $(q(X), \{\{q(a)\}, \{q(X)\}\}, \{0, 0, 1\}, 0)$,
$TB(r)$: $(r, \{\{\}, \{r\}\}, \{0\}, 1)$,
$TB(s)$: $(s, \{\{\}, \{s\}\}, \{0\}, 1)$,
$TB(w)$: $(w, \{\{\}, \{\}\}, \{0\}, 1)$. □

From Fig. 1 we observe that each node in the tree has a unique name (index) N_i that is labeled by a goal G_i, so that the left-most subgoal $A_1 = A$ (or $A_1 = \neg A$) of G_i is uniquely determined by N_i. In order to keep track of A_1 that resolves against both program clauses and tabled answers, we attach to N_i

three pointers. $N_i \rightarrow t\_answer\_ptr$ and $N_i \rightarrow u\_answer\_ptr$ point to an answer in $TB(A) \rightarrow t\_answer$ and $TB(A) \rightarrow u\_answer$, respectively. $N_i \rightarrow clause\_ptr$ points to a clause whose head is unifiable with A. This leads to the following.

Definition 1. *Let G_i be a goal $\leftarrow A_1, ..., A_m$ $(m \geq 1)$. By* **"register a node N_i with G_i"** *we do the following: (1) label N_i with G_i; (2) create the above three pointers for N_i, which unless otherwise specified are initialized to null.*

We assume two table functions: *memo*(.) and *lookup*(.). Let N_i be a node with the left-most subgoal A. Let I be an answer of A with truth type $S \in \{t, u, u^*\}$. When $TB(A)$ contains no answer with truth value t that is a variant of or more general than I, $memo(N_i, I, S)$ adds I to $TB(A)$ in the following way. When $S = t$, add I to the end of $TB(A) \rightarrow t\_answer$, set $TB(A) \rightarrow COMP = 1$ if I is a variant of A, and remove from $TB(A) \rightarrow u\_answer$ all J/J^* with J an instance/variant of I. Otherwise, if $S = u$ (resp. $S = u^*$), add I (resp. I^*) to the end of $TB(A) \rightarrow u\_answer$ provided that it contains no answer that is a variant of or more general than I (resp. I^*).

Let N_i and A be as above, and I and S be variables that are used for caching an answer and its truth type. $lookup(N_i, I, S)$ fetches from $TB(A)$ an answer with its truth type into I and S, respectively. If no answer is available in $TB(A)$, $I = null$.

3.2 Resolvants

We now discuss how to resolve subgoals against program clauses as well as tabled answers. Let N_i be a node labeled by a goal $G_i = \leftarrow A_1, ..., A_m$ $(m \geq 1)$ with $A_1 = p(\boldsymbol{X})$. Consider evaluating A_1 using a program clause $C = A \leftarrow B_1, ..., B_n$ $(n \geq 0)$, where $A_1\theta = A\theta$.[3] In Prolog, we will generate 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 θ is *consumed* by all A_js $(j > 1)$, although the proof of $A_1\theta$ has not yet been completed (*produced*). In our tabulated resolution, however, we apply the *PMF* (for Prove-Memorize-Fetch) mode to resolve subgoals against clauses and tabled answers [11]. That is, we first prove $(B_1, ..., B_n)\theta$. If it is true with some mgu θ_1, which means $A_1\theta\theta_1$ is true, we memorize the answer in the table $TB(A_1)$ if it is new. We then fetch an answer $p(\boldsymbol{I})$ with truth type S from $TB(A_1)$ and apply it to the remaining subgoals of G_i. The process can be depicted more clearly in Fig. 2.

Obviously the PMF mode preserves the original set of answers of A_1. Moreover, since only new answers of A_1 are added to the table, all repeated answers of A_1 will be precluded to apply to the remaining subgoals of G_i.

The PMF mode can readily be realized by using the two table procedures *memo*(.) and *lookup*(.). That is, after resolving the subgoal A_1 with the clause C, N_i gives a child node N_{i+1} labeled with the goal $G_{i+1} = \leftarrow (B_1, ..., B_n)\theta$, $memo(N_i, p(\boldsymbol{X})\theta, t)$, $lookup(N_i, I_i, S_i)$, $A_2, ..., A_m$. Note that the propagation of

[3] Here and throughout, we assume that C has been standardized apart to share no variables with G_i.

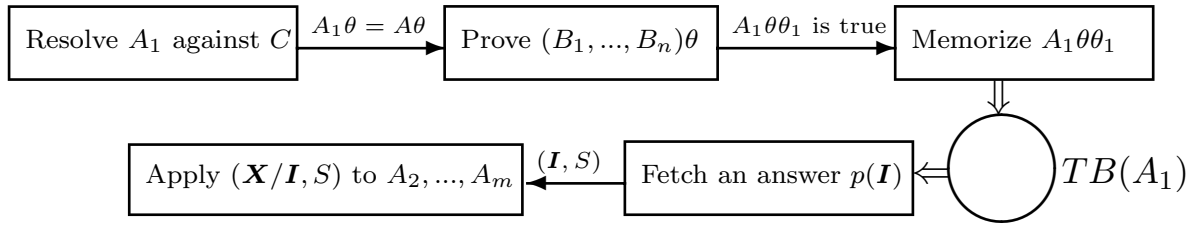

Fig. 2. The PMF mode for resolving subgoals.

θ is blocked by the subgoal $lookup(N_i, I_i, S_i)$ because the consumption (fetch) must be after the production (prove and memorize).

Observe that after the proof of A_1 is reduced to the proof of $(B_1, ..., B_n)\theta$, $memo(N_i, p(\boldsymbol{X})\theta, t)$, $lookup(N_i, I_i, S_i)$ by applying a program clause C, the truth value of an answer of A_1 to be memorized must be the logical AND of the truth values of the answers of all $B_j\theta$s. Such an AND computation is carried out incrementally. Initially we have $memo(N_i, p(\boldsymbol{X})\theta, S')$ with $S' = t$. Then from $j = 1$ to $j = n$ if $B_j\theta$ gets an answer $B_j\theta\theta'$with truth type S, the $memo(.)$ subgoal is updated to $memo(N_i, p(\boldsymbol{X})\theta\theta', S' \wedge S)$. This leads to the following definition.

Definition 2. *Let $G_1 = \leftarrow A_1, ..., A_m$ be a goal, θ an mgu, and $S \in \{t, u, u^*\}$. The* **resultant** *of applying (θ, S) to G_1 is the goal $G_2 = \leftarrow (A_1, ..., A_{k-1})\theta, A'_k\theta$, A_{k+1}, ..., A_m, where A_k is the left-most subgoal of the form $memo(.)$ (if G_1 contains no $memo(.)$, $k = m$) and A'_k is A_k with its answer type $S' \in \{t, u, u^*\}$ changed to $S' \wedge S$.*

The concept of resolvants of TPWF-resolution is then defined based on the PMF mode.

Definition 3. *Let N_i be a node labeled by a goal $G_i = \leftarrow A_1, ..., A_m$ $(m \geq 1)$.*

1. *If $A_1 = p(\boldsymbol{X})$, let C be a clause $A \leftarrow B_1, ..., B_n$ with $A\theta = A_1\theta$, then*
   a) *The* **resolvant** *of G_i and C is the goal $G_{i+1} = \leftarrow (B_1, ..., B_n)\theta, memo(N_i, p(\boldsymbol{X})\theta, t), lookup(N_i, I_i, S_i), A_2, ..., A_m$.*
   b) *Let $p(\boldsymbol{I})$ be an answer of A_1 with truth type S, then the* **resolvant** *of G_i and $p(\boldsymbol{I})$ with S is the resultant of applying $(\boldsymbol{X}/\boldsymbol{I}, S)$ to $\leftarrow A_2, ..., A_m$.*
2. *If $A_1 = \neg B$ with B a ground atom, let B be the answer with truth type $S \in \{f, u, u^*\}$, then the* **resolvant** *of G_i and B with S is the resultant of applying $(\{\}, \neg S)$ to $\leftarrow A_2, ..., A_m$.*
3. *If A_1 is $memo(N_h, q(\boldsymbol{I}), S)$ and A_2 is $lookup(N_h, I_h, S_h)$, let $q(\boldsymbol{X})$ be the left-most subgoal at node N_h, then (after executing the two functions) the* **resolvant** *of G_i and I_h with truth type S_h is the resultant of applying $(\boldsymbol{X}/I_h, S_h)$ to $\leftarrow A_3, ..., A_m$.*

3.3 Ancestor Lists and Loops

Loop checking is a principal feature of TPWF-resolution (see Example 1). Positive and negative loops are determined based on ancestor lists that are associated with subgoals.

Definition 4. *([10] with slight modification) An* **ancestor list** AL_A *is associated with each subgoal* A *in a tree (see the TPWF-tree below), which is defined recursively as follows.*

1. *If* A *is at the root, then* $AL_A = \{\}$.
2. *Let* A *be at node* N_{i+1}. *If* A *inherits a subgoal* A' *(by copying or instantiation) from its parent node* N_i, *then* $AL_A = AL_{A'}$; *else if* A *is in the resolvant of a subgoal* B *at node* N_i *and a clause* $B' \leftarrow A_1, ..., A_n$ *with* $B\theta = B'\theta$ *(i.e.* $A = A_i\theta$ *for some* $1 \leq i \leq n$*),* $AL_A = \{(N_i, B)\} \cup AL_B$.
3. *Let* $G_i = \leftarrow \neg A, ...$ *be the goal at* N_i, *which has a child node* N_{i+1} *labeled by the goal* $G_{i+1} = \leftarrow A$ *(the edge from* N_i *to* N_{i+1} *is dotted; see Fig. 1). Then* $AL_A = \{\neg\} \cup AL_{\neg A}$.

Let G_i at node N_i and G_k at node N_k be two goals in a derivation and A and A' be the left-most subgoals of G_i and G_k, respectively. If A is in the ancestor list of A', i.e. $(N_i, A) \in AL_{A'}$, the proof of A needs the proof of A'. In such a case, we call A (resp. N_i) an *ancestor subgoal* of A' (resp. *ancestor node* of N_k). Particularly, if A is both an ancestor subgoal and a variant, i.e. an *ancestor variant subgoal*, of A', we say the derivation goes into a *loop*. The loop is *negative* if there is a $\neg$ ahead of (N_i, A) in $AL_{A'}$; otherwise, it is *positive*.

For example, the ancestor list of the subgoal r at N_7 in Fig. 1 is

$$AL_r = \{\neg, (N_5, s), \neg, (N_3, r), \neg, (N_1, q(X)), (N_0, p(X))\}$$

and the ancestor list of the subgoal $p(X)$ at N_{12} is

$$AL_{p(X)} = \{(N_1, q(X)), (N_0, p(X))\}.$$

There is a negative loop between N_3 and N_7, and a positive loop between N_0 and N_{12}.

3.4 Control Strategy

Although in principle the tabulated approach presented in this paper is effective for any fixed control strategy, we choose to use the so called TP-strategy, which is the Prolog control strategy enhanced with mechanisms for selecting tabled answers.

Definition 5 ([11]). *By* 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 tabled answers and program clauses) + Last-first (for backtracking).*

3.5 Algorithm for Building TPWF-Trees

In order to simplify the presentation, we assume every subgoal has a table and that the flag variables $COMP$ (in tables) and NEW are updated automatically. Moreover, we assume that whenever an atom A is *assumed* undefined (i.e. $A = u^*$ is assumed), A is added to UA, and that whenever $A = u^*$ is *derived* (memorized), A is added to UD (automatically). We assume that when selecting clauses to resolve with subgoals, all clauses whose status is *"no longer available"* are automatically skipped. Finally we assume a function $return(A, S)$, which returns an answer A with truth type S. The truth type of $return(A, S)$ is updated in the same way as $memo(\_, \_, S)$.

TPWF-trees are constructed based on TP-strategy using the following algorithm.

Definition 6 (TPWF-Algorithm). *Let P be a logic program, A an atom, and $G_0 = \leftarrow A, return(A, t)$. Let $AL_A = \{\}$ be the ancestor list of A. The* TPWF-tree *TF_{G_0} of $P \cup \{G_0\}$ is constructed by applying the following algorithm until the answer NO or* FLOUND *is returned.*

$tpwf(G_0, AL_A)$:

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

Case 2.1: A_1 is $return(A', S)$. Return (A', S) if $S \neq u^*$. When $S = t$ or $S = u$, set $G_{i+1} = \mathcal{T}$ and $G_{i+1} = \mathcal{U}$, respectively. Goto 3 with $N = N_i$.

Case 2.2: A_1 is $memo(N_h, I, S)$ and A_2 is $lookup(N_h, I_h, S_h)$. Execute the two table functions. If $I_h = null$, then goto 3 with $N = N_i$; else set G_{i+1} to the resolvant of G_i and I_h with truth type S_h and goto 2.

Case 2.3: $A_1 = \neg B$. If B is non-ground, set $G_{i+1} = \mathcal{FD}$ and return FLOUND*. Get an answer from $TB(B)$. Let I be the answer with truth type S.*

Case 2.3.1: $I \neq null$. If $S = t$, then goto 3 with $N = N_i$; else set G_{i+1} to the resolvant of G_i and I with S and goto 2.

Case 2.3.2: $I = null$. When $TB(B) \rightarrow COMP = 1$, if $TB(B) \rightarrow u\_answer \neq \{\}$, then goto 3 with $N = N_i$; else set $G_{i+1} = \leftarrow A_2, ..., A_m$ and goto 2. Otherwise, let $G'_0 = \leftarrow B, return(B, t)$ and $AL_B = \{\neg\} \cup AL_{A_1}$. Call $tpwf(G'_0, AL_B)$ until NO or $FLOUND$ is returned. If $FLOUND$ is returned, then set $G_{i+1} = \mathcal{FD}$ and return $FLOUND$; else apply the answers in $TB(B)$ to A_1 (repeat Case 2.3) and then goto 3 with $N = N_i$.

Case 2.4: $A_1 = p(\boldsymbol{X})$. Get an answer I with truth type S from $TB(A_1)$. If $I \neq null$, then set G_{i+1} to the resolvant of G_i and I with S and goto 2; else

Case 2.4.1: $TB(A_1) \rightarrow COMP = 1$. Goto 3 with $N = N_i$.

Case 2.4.2: N_i is a top loop node. Do answer iteration and then goto 3 with $N = N_i$.

Case 2.4.3: $A_1 \in UA$. if A_1^* is in $TB(A_1) \rightarrow u\_answer$, then goto 3 with $N = N_i$; else set G_{i+1} to the resolvant of G_i and A_1 with truth type u^*, and goto 2.[4]

Case 2.4.4: Otherwise. If no loop occurs (i.e. A_1 has no ancestor variant subgoal), then resolve A_1 with the first clause available; else resolve A_1 with the first clause below the one that is being used by its closest ancestor variant subgoal. If such a clause C_{p_j} exists, then set G_{i+1} to the resolvant of G_i and C_{p_j} and goto 2; else goto 3 with $N = N_i$ while assuming $A_1 = u^*$ if the loop is negative.

3. *Backtracking: If N is the root, return NO. Let N_f be the parent node of N with the left-most subgoal A_f. If A_f is a function, goto 3 with $N = N_f$. Otherwise, if N was generated from N_f by resolving A_f with a clause C_j, then if A_f is not involved in any loop, set $TB(A_f) \rightarrow clause\_status[j] = 0$. Goto 2 with N_f as the latest registered node.*

The input of TPWF-Algorithm includes a top goal $G_0 = \leftarrow A, return(A, t)$ and an ancestor list AL_A. Its output is either $FLOUND$, indicating that G_0 is floundered, or NO, showing that there is no more answer for A, or (A', S), meaning that A' is an answer of A with truth type $S \in \{t, u\}$.

Observe that like SLDNF-resolution [4], when $A_1 = \neg B$ we may build a new tree for B (Case 2.3.2). In SLDNF-resolution, the two SLDNF-trees are totally independent. This leads to possible infinite negative loops. TPWF-resolution, however, connects the two TPWF-trees via the ancestor list $\{\neg\} \cup AL_{A_1}$, so that negative loops can be detected effectively (see Fig. 1).

4 TPWF-Resolution

Definition 7. *Let TF_{G_0} be a TPWF-tree of $P \cup \{G_0\}$. All leaves of TF_{G_0} labeled by $\mathcal{T}$, $\mathcal{U}$ or $\mathcal{FD}$ are* **success**, **undefined** *and* **flounder** *leaves, respectively, and all other leaves are* **failure** *leaves. A* **TPWF-derivation** *is a partial branch in TF_{G_0} starting at the root, which is* **successful**, **floundered**, **undefined** *or* **failed** *if it ends respectively with a success leaf, a flounder leaf, an undefined leaf and a failure leaf. The process of constructing TPWF-derivations is called* **TPWF-resolution**.

A goal G_0 is floundered if it has a floundered TPWF-derivation. Let G_0 be a non-floundered goal and I' be a variant of or more general than I. Then G_0 is **true with an answer** *I if there is a successful TPWF-derivation with (I', t) returned;* **undefined with** *I if it is not true with any instance of I but there is an undefined TPWF-derivation with (I', u) returned;* **false with** *I if it is neither true nor undefined with any instance of I.*

The following theorem follows from the basic fact: For any logic program with the bounded-term-size property, (1) the set of answers in any table is finite, (2) every TPWF-derivation is finite, and (3) answer iteration must reach a fixpoint.

[4] For this case, no further backtracking will be allowed at this node.

Theorem 1 (Termination of TPWF-resolution). *Let P be a logic program with the bounded-term-size property and $G_0 = \leftarrow A, return(A,t)$ a top goal. TPWF-Algorithm terminates with a finite TPWF-tree.*

TPWF-resolution cuts infinite loops and infinite recursions through negation by means of assumption and answer iteration. Positive loops are cut simply by backtracking, whereas negative loop subgoals whose truth values are currently undecided will be assumed temporarily undefined before being failed via backtracking. Temporarily undefined values will be removed (from tables) after their t or u counterparts are derived. This guarantees the correctness of loop cutting. Meanwhile, before leaving a loop by failing its top loop node, iteration will be carried out to derive complete answers of loop subgoals. For logic programs with the bounded-term-size property, the iteration must terminate with a fixpoint of answers. This leads to the following.

Theorem 2 (Soundness and Completeness of TPWF-resolution). *Let P be a logic program with the bounded-term-size property and $G_0 = \leftarrow A, return(A,t)$ a non-floundered goal. Let $WF(P)$ be the well-founded model of P. Then*

1. *$WF(P) \models \exists(A)$ iff G_0 is true with an instance of A;*
2. *$WF(P) \models \neg\exists(A)$ iff G_0 is false with A;*
3. *$WF(P) \models \forall(A\theta)$ iff G_0 is true with $A\theta$;*
4. *$WF(P) \not\models \exists(A)$ and $WF(P) \not\models \neg\exists(A)$ iff G_0 is undefined with A.*

Acknowledgments We thank the anonymous referees for their helpful comments. The first author is supported in part by Chinese National Natural Science Foundation and Trans-Century Training Programme Foundation for the Talents by the Chinese Ministry of Education.

References

1. Bol, R. N., Degerstedt, L.: Tabulated Resolution for the Well-Founded Semantics. *Journal of Logic Programming* 34:2 (1998) 67-109
2. Chen, W. D., Swift, T., Warren, D. S.: Efficient Top-Down Computation of Queries under the Well-Founded Semantics. *Journal of Logic Programming* 24:3 (1995) 161-199
3. Chen, W. D., Warren, D. S.: Tabled Evaluation with Delaying for General Logic Programs. *J. ACM* 43:1 (1996) 20-74
4. Lloyd, J. W.: *Foundations of Logic Programming.* 2nd edn. Springer-Verlag, Berlin (1987)
5. Przymusunski, T.: Every Logic Program Has a Natural Stratification and an Iterated Fixed Point Model. In: *Proc. of the 8th ACM Symposium on Principles of Database Systems* (1989) 11-21
6. Ross, K.: A Procedural Semantics for Well-Founded Negation in Logic Programs. *Journal of Logic Programming* 13:1 (1992) 1-22
7. Sagonas, K., Swift, T., Warren, D. S.: XSB as an Efficient Deductive Database Engine. In: *Proc. of the ACM SIGMOD Conference on Management of Data.* Minneapolis (1994) 442-453

8. Sagonas, K., Swift, T., Warren, D. S.: An Abstract Machine for Tabled Execution of Fixed-Order Stratified Logic Programs.*ACM Transactions on Programming Languages and Systems* 20:3 (1998)
9. Sagonas, K., Swift, T., Warren, D. S., Freire, J., Rao, P.: *The XSB Programmer's Manual* (Version 1.8) (1998)
10. Shen, Y. D.: An Extended Variant of Atoms Loop Check for Positive Logic Programs.*New Generation Computing* 15:2 (1997) 317-341
11. Shen, Y. D., Yuan, L. Y., You, J. H., Zhou, N. F.: Linear Tabulated Resolution Based on Prolog Control Strategy. Submitted for publication (1999)
12. Van Gelder, A., Ross, K., Schlipf, J.: The Well-Founded Semantics for General Logic Programs. *J. ACM* 38:3 (1991) 620-650
13. Van Gelder, A.: Negation as Failure Using Tight Derivations for General Logic Programs. *Journal of Logic Programming* 6:1&2 (1989) 109-133
14. Warren, D. H. D.: An Abstract Prolog Instruction Set. Technical Report 309, SRI International (1983)
15. Zhou, N. F.: Parameter Passing and Control Stack Management in Prolog Implementation Revisited. *ACM Transactions on Programming Languages and Systems* 18:6 (1996) 752-779