Enhancing Global SLS-Resolution with Loop Cutting and Tabling Mechanisms

Yi-Dong Shen

Laboratory of Computer Science, Institute of Software

Chinese Academy of Sciences, Beijing 100080, China

Email: ydshen@ios.ac.cn

Jia-Huai You and Li-Yan Yuan

Department of Computing Science, University of Alberta

Edmonton, Alberta, Canada T6G 2H1

Email: {you, yuan}@cs.ualberta.ca

Abstract

Global SLS-resolution is a well-known procedural semantics for top-down computation of queries under the well-founded model. It inherits from SLDNF-resolution the *linearity* property of derivations, which makes it easy and efficient to implement using a simple stack-based memory structure. However, like SLDNF-resolution it suffers from the problem of infinite loops and redundant computations. To resolve this problem, in this paper we develop a new procedural semantics, called *SLTNF-resolution*, by enhancing Global SLS-resolution with loop cutting and tabling mechanisms. SLTNF-resolution is sound and complete w.r.t. the well-founded semantics for logic programs with the bounded-term-size property, and is superior to existing linear tabling procedural semantics such as SLT-resolution.

Keywords: Logic programming, the well-founded semantics, Global SLS-resolution, loop cutting, tabling.

1 Introduction

There are two types of semantics for a logic program: a declarative semantics and a procedural semantics. The declarative semantics formally defines the meaning of a logic program by specifying an *intended model* among all models of the logic program, whereas the procedural semantics implements/computes the declarative semantics by providing an algorithm for

evaluating queries against the logic program. Most existing procedural semantics are built upon the well-known *resolution rule* created by Robinson [20].

Prolog is the first yet the most popular logic programming language [13]. It adopts *SLDNF-resolution* as its procedural semantics [9]. One of the best-known properties of SLDNF-resolution is its *linearity* of derivations, i.e., its query evaluation (i.e., SLDNF-derivations) constitutes a search tree, called an *SLDNF-tree*, which can be implemented easily and efficiently using a simple stack-based memory structure [33, 35]. However, SLDNF-resolution suffers from two serious problems. First, its corresponding declarative semantics, i.e. the *predicate completion semantics* [9], is based on two truth values (either *true* or *false*) and thus incurs inconsistency for some logic programs like $P = \{p(a) \leftarrow \neg p(a)\}$ [14, 27]. Second, it may generate infinite loops and a large amount of redundant sub-derivations [2, 10, 24].

To overcome the first problem with SLDNF-resolution, the *well-founded semantics* [32] is introduced as an alternative to the predicate completion semantics. A well-founded model accommodates three truth values: *true, false* and *undefined*, so that inconsistency is avoided by letting atoms that are recursively connected through negation undefined. Several procedural semantics have been developed as an alternative to SLDNF-resolution to compute the well-founded semantics, among the most representative of which are *Global SLS-resolution* [17, 21] and *SLG-resolution* [7, 8, 3].

Global SLS-resolution is a direct extension of SLDNF-resolution. It evaluates queries under the well-founded semantics by generating a search tree, called an *SLS-tree*, in the same way as SLDNF-resolution does except that infinite derivations are treated as *failed* and infinite recursions through negation as *undefined*. Global SLS-resolution retains the linearity property of SLDNF-resolution, but it also inherits the problem of infinite loops and redundant computations. Moreover, Global SLS-resolution handles negation as follows: A ground atom A is false when all branches of the SLS-tree for A are either infinite or end at a failure leaf. Infinite branches make Global SLS-resolution not effective in general [21].

To resolve infinite loops and redundant computations, the *tabling* technique is introduced [29, 34]. The main idea of tabling is to store intermediate answers of subgoals and then apply them to solve variants of the subgoals. With tabling no variant subgoals will be recomputed by applying the same set of clauses, so infinite loops can be avoided and redundant computations be substantially reduced. There are two typical ways to make use of tabling to compute the well-founded semantics. One is to directly enhance SLDNF-resolution or Global SLS-resolution with tabling while the other is to create a new tabling mechanism with a different derivation structure. SLG-resolution results from the second way [3, 8]. Due to the use of tabling, SLG-resolution gets rid of infinite loops and reduces redundant computations. However, it does not have the linearity property since its query evaluation constitutes a search forest instead of a search tree. As a result, it cannot be implemented in the same way as SLDNF-resolution using a simple stack-based memory structure [22, 23, 28].

In [26] an attempt is made to directly enhance SLDNF-resolution with tabling to compute the well-founded semantics, which leads to a tabling mechanism, called *SLT-resolution*. SLT-resolution retains the linearity property, thus is referred to as a *linear tabling* mechanism. Due to the use of tabling, it is free of infinite loops and has fewer redundant computations than SLDNF-resolution. However, SLT-resolution has the following two major drawbacks: (1) It defines positive loops and negative loops based on the same ancestor-descendant relation, which makes loop detection and handling quite costly since a loop may go across several (subsidiary) SLT-trees. (2) It makes use of answer iteration to derive all answers of looping subgoals, but provides no answer completion criteria for pruning redundant derivations. Note that answer completion is the key to an efficient tabling mechanism.

In this paper, we develop a new procedural semantics, called *SLTNF-resolution*, for the well-founded semantics by enhancing Global SLS-resolution with tabling and loop cutting mechanisms. SLTNF-resolution retains the linearity property and makes use of tabling to get rid of all loops and reduce redundant computations. It defines positive and negative loops in terms of two different ancestor-descendant relations, one on subgoals within an SLS-tree and the other on SLS-trees, so that positive and negative loops can be efficiently detected and handled. It employs two effective criteria for answer completion of tabled subgoals so that redundant derivations can be pruned as early as possible. All these mechanisms are integrated into an algorithm quite like that for generating SLS-trees.

The paper is organized as follows. Section 2 reviews Global SLS-resolution. Section 3 defines ancestor-descendant relations for identifying positive and negative loops, develops an algorithm for generating SLTNF-trees, establishes criteria for determining answer completion of tabled subgoals, and proves the correctness of SLTNF-resolution. Section 4 mentions some related work, and Section 5 concludes.

2 Preliminaries and Global SLS-Resolution

In this section, we review some standard terminology of logic programs [14] and recall the definition of Global SLS-Resolution. We do not repeat the definition of the well-founded model here; it can be found in [32, 17, 19] and many other papers.

Variables begin with a capital letter, and predicate, function and constant symbols with a lower case letter. By a *variant* of a literal L we mean a literal L' that is identical to L up to variable renaming.

Definition 2.1 A *general logic program* (logic program for short) is a finite set of clauses of the form

$$A \leftarrow L_1, ..., L_n$$

where A is an atom and L_is are literals. A is called the *head* and $L_1, ..., L_n$ is called the *body* of the clause. When $n = 0$, the "$\leftarrow$" symbol is omitted. If a logic program has no clause

with negative literals in its body, it is called a *positive logic program.*

Definition 2.2 A *goal* G is a headless clause $\leftarrow L_1, ..., L_n$ where each L_i is called a *subgoal.* A goal is also written as $G = \leftarrow Q$ where $Q = L_1, ..., L_n$ is called a *query.* A *computation rule* (or *selection rule*) is a rule for selecting one subgoal from a goal.

Let $G_i = \leftarrow L_1, ..., L_j, ..., L_n$ be a goal with L_j a positive subgoal. Let $C = L \leftarrow F_1, ..., F_m$ be a clause such that L and L_j are unifiable, i.e. $L\theta = L_j\theta$ where θ is an mgu (most general unifier). The *resolvent* of G_i and C on L_j is a goal $G_k = \leftarrow (L_1, ..., L_{j-1}, F_1, ..., F_m, L_{j+1}, ..., L_n)\theta$. In this case, we say that the proof of G_i is reduced to the proof of G_k.

The initial goal, $G_0 = \leftarrow L_1, ..., L_n$, is called a *top* goal. Without loss of generality, we shall assume throughout the paper that a top goal consists only of one atom (i.e. $n = 1$ and L_1 is a positive literal).

Trees are used to depict the search space of a top-down query evaluation procedure. For convenience, a node in such a tree is represented by $N_i : G_i$ where N_i is the node name and G_i is a goal labeling the node. Assume no two nodes have the same name, so we can refer to nodes by their names.

Let P be a logic program and $G_0 = \leftarrow Q$ a top goal. Global SLS-resolution is the process of constructing SLS-derivations from $P \cup \{G_0\}$ via a computation rule R. An *SLS-derivation* is a partial branch beginning at the root $N_0 : G_0$ of an SLS-tree. Every leaf of an SLS-tree is either a *success* leaf or a *failure* leaf or a *flounder* leaf or an *undefined* leaf.[1] Q is a *non-floundering query* if no SLS-tree for evaluating Q under R contains a flounder leaf.

An SLS-tree is *successful* if it has a success leaf. It is *failed* if all of its branches are either infinite or end at a failure leaf. It is *floundered* if it contains a floundered leaf and is not successful. An SLS-tree is *undefined* if it is neither successful nor failed nor floundered.

There are two slightly different definitions of an SLS-tree: Przymusinski's definition [17, 18] and Ross' definition [21]. Przymusinski's definition requires a level mapping (called *strata*) to be associated with literals and goals, while Ross' definition requires the computation rule to be *preferential*, i.e. positive subgoals are selected ahead of negative ones and negative subgoals are selected in parallel. Both of the two definitions allow infinite branches and infinite recursion through negation. The following definition of an SLS-tree is obtained by combining the two definitions.

Definition 2.3 (SLS-trees [17, 18, 21]) Let P be a logic program, G_0 a top goal, and R a computation rule. The *SLS-tree* $T_{N_0:G_0}$ for $P \cup \{G_0\}$ via R is a tree rooted at $N_0 : G_0$ such that for any node $N_i : G_i$ in the tree with $G_i = \leftarrow L_1, ..., L_n$:

1. If $n = 0$ then N_i is a *success* leaf, marked by $\Box_t$.

[1] In [18], an undefined leaf is called a *non-labeled* leaf.

2. If L_j is a positive literal selected by R, then for each clause C in P whose head is unifiable with L_j, N_i has a child $N_k : G_k$ where G_k is the resolvent of C and G_i on L_j. If no such a clause exists in P, then N_i is a *failure* leaf, marked by $\Box_f$.

3. Let $L_j = \neg A$ be a negative literal selected by R. If A is not ground then N_i is a *flounder* leaf, marked by $\Box_{fl}$, else let $T_{N_{i+1}:\leftarrow A}$ be an (subsidiary) SLS-tree for $P \cup \{\leftarrow A\}$ via R. We consider four cases:

   (a) If $T_{N_{i+1}:\leftarrow A}$ is failed then N_i has only one child that is labeled by the goal $\leftarrow L_1, ..., L_{j-1}, L_{j+1}, ..., L_n$.

   (b) If $T_{N_{i+1}:\leftarrow A}$ is successful then N_i is a *failure* leaf, marked by $\Box_f$.

   (c) If $T_{N_{i+1}:\leftarrow A}$ is floundered then N_i is a *flounder* leaf, marked by $\Box_{fl}$.

   (d) Otherwise (i.e. $T_{N_{i+1}:\leftarrow A}$ is undefined), we mark L_j in G_i as *skipped* and use the computation rule R to select a new literal L_k from G_i and apply the resolution steps 2 and 3 to the goal G_i. If all literals in G_i were already marked as skipped then N_i is an *undefined* leaf, marked by $\Box_u$.

We make two remarks. First, the level mapping/strata used in Przymusinski's definition is implicit in Definition 2.3. That is, in case 3 the level/stratum of A is less than the level/stratum of G_i if and only if either case 3a or case 3b or case 3c holds. Second, the preferential restriction of Ross' definition to the computation rule is relaxed by marking undefined subgoals as skipped and then continuing to select new subgoals from the remaining subgoals in G_i for evaluation (see case 3d). A leaf is undefined if and only if all its subgoals are marked as skipped.

Definition 2.4 A *successful* (resp. *failed* or *undefined*) derivation for a goal G is a branch beginning at the root of the SLS-tree for G and ending at a success (resp. failure or undefined) leaf. A *correct answer substitution* for G is the substitution $\theta = \theta_1...\theta_n$, where θ_is are the most general unifiers used at each step along the derivation, restricted to the variables in G.

It has been shown that Global SLS-resolution is sound and complete with respect to the well-founded semantics for non-floundering queries.

Theorem 2.1 ([17, 18, 21]) *Let P be a logic program, R a computation rule, and $G_0 \leftarrow Q$ be a top goal with Q a non-floundering query under R. Let $WF(P)$ be the well-founded model of P.*

1. *$WF(P) \models \exists(Q)$ if and only if the SLS-tree for $P \cup \{G_0\}$ via R is successful.*

2. *$WF(P) \models \forall(Q\theta)$ if and only if there exists a correct answer substitution for G_0 more general than the substitution θ.*

3. $WF(P) \models \neg\exists(Q)$ if and only if the SLS-tree for $P \cup \{G_0\}$ is failed.

Definition 2.5 Let $N_i : G_i$ be a node in an SLS-tree $T_{N_r:G_r}$ where A is the selected positive subgoal in G_i. The partial branches of $T_{N_r:G_r}$ beginning at N_i that are used to evaluate A constitute *sub-derivations* for A. All such sub-derivations form a *sub-SLS-tree* for A at N_i.

By Theorem 2.1, for any correct answer substitution θ built from a successful sub-derivation for A, $WF(P) \models \forall(A\theta)$.

Since Global SLS-resolution allows infinite derivations as well as infinite recursion through negation, we may need infinite time to generate an SLS-tree. This is not feasible in practice. In the next section, we resolve this problem by enhancing Global SLS-resolution with both loop cutting and tabling mechanisms.

3 SLTNF-Resolution

We first define an ancestor-descendant relation on selected subgoals in an SLS-tree. Informally, A is an *ancestor subgoal* of B if the proof of A depends on (or in other words goes via) the proof of B. For example, let $M :\leftarrow A, A_1, ..., A_m$ be a node in an SLS-tree, and $N :\leftarrow B_1\theta, ..., B_n\theta, A_1\theta, ..., A_m\theta$ be a child node of M that is generated by resolving M on the subgoal A with a clause $A' \leftarrow B_1, ..., B_n$ where $A\theta = A'\theta$. Then A at M is an ancestor subgoal of all $B_i\theta$s at N. However, such relationship does not exist between A at M and any $A_j\theta$ at N. It is easily seen that all $B_i\theta$s at N inherit the ancestor subgoals of A at M, and that each $A_j\theta$ at N inherits the ancestor subgoals of A_j at M. Note that subgoals at the root of an SLS-tree have no ancestor subgoals.

Let $N_i : G_i$ and $N_k : G_k$ be two nodes and A and B be the selected subgoals in G_i and G_k, respectively. When A is an ancestor subgoal of B, we refer to B as a *descendant subgoal* of A, N_i as an *ancestor node* of N_k, and N_k as a *descendant node* of N_i. Particularly, if A is both an ancestor subgoal and a variant, i.e. an *ancestor variant subgoal*, of B, we say the derivation goes into a *loop*, where N_i and N_k are respectively called an *ancestor loop node* and a *descendant loop node*, and A (at N_i) and B (at N_k) are respectively called an *ancestor loop subgoal* and a *descendant loop subgoal*.

The above ancestor-descendant relation is defined over subgoals and will be applied to detect positive loops, i.e. loops within an SLS-tree. In order to handle negative loops (i.e. loops through negation like $A \leftarrow \neg B$ and $B \leftarrow \neg A$) which occur across SLS-trees, we define an ancestor-descendant relation on SLS-trees. Let $N_i :\leftarrow \neg A, ...$ be a node in $T_{N_r:G_r}$, with $\neg A$ the selected subgoal, and let $T_{N_{i+1}:\leftarrow A}$ be an (subsidiary) SLS-tree for $P \cup \{\leftarrow A\}$ via R. $T_{N_r:G_r}$ is called an *ancestor SLS-tree* of $T_{N_{i+1}:\leftarrow A}$, while $T_{N_{i+1}:\leftarrow A}$ is called a *descendant SLS-tree* of $T_{N_r:G_r}$. Of course, the ancestor-descendant relation is transitive.

A negative loop occurs if an SLS-tree has a descendant SLS-tree, with the same goal at their roots. For convenience, we use dotted edges to connect parent and child SLS-trees, so

that negative loops can be clearly identified. Let G_0 be a top goal. We call $T_{N_0:G_0}$ together with all of its descendant SLS-trees a *generalized SLS-tree*, denoted GT_{P,G_0} (or simply GT_{G_0} when no confusion would arise). Therefore, a branch of a generalized SLS-tree may come across several SLS-trees through dotted edges. A *generalized SLS-derivation* is a partial branch beginning at the root of a generalized SLS-tree.

Assume that all loops are detected and cut based on the ancestor-descendant relations. This helps Global SLS-resolution get rid of infinite derivations and infinite recursion through negation. However, applying such loop cutting mechanism alone is not effective since some answers would be lost. In order to guarantee the completeness of Global SLS-resolution with the loop cutting mechanism, we introduce a tabling mechanism into SLS-derivations, leading to a tabulated SLS-resolution.

In tabulated resolutions, the set of predicate symbols in a logic program is partitioned into two groups: *tabled predicate symbols* and *non-tabled predicate symbols*. Subgoals with tabled predicate symbols are then called *tabled subgoals*. A *dependency graph* [1] is used to make such classification. Informally, for any predicate symbols p and q, there is an edge $p \rightarrow q$ in the dependency graph G_P of a logic program P if and only if P contains a clause whose head contains p and whose body contains q. p is a tabled predicate symbol if G_P contains a cycle involving p. It is trivial to show that subgoals involved in any loops in SLS-trees must be tabled subgoals.

Intermediate answers of tabled subgoals will be stored in tables once they are produced at some derivation stages. Such answers are called *tabled answers*. For convenience of presentation, we organize a table into a compound structure like *struct* in pseudo C^{++} language. That is, the table of an atom A, denoted TB_A, is internally an instance of the data type TABLE defined as follows:

```
typedef struct {
    string    atom; //for TB_A, atom = A.
    int       comp; //status of atom indicating if all answers have been tabled.
    set       ans; //tabled answers of atom.
} TABLE;
```

Answers of a tabled subgoal A are stored in $TB_A \rightarrow ans$. We say TB_A is *complete* if $TB_A \rightarrow ans$ contains all answers of A. We use $TB_A \rightarrow comp = 1$ to mark the completeness of tabled answers. Clearly, the case $TB_A \rightarrow comp = 1$ and $TB_A \rightarrow ans = \emptyset$ indicates that A is false.

We introduce a special subgoal, u^*, which is assumed to occur neither in logic programs nor in top goals. u^* will be used to substitute for some ground negative subgoals whose truth values are temporarily undefined (i.e., whether they are true or false cannot be determined at the current stage of derivation). We assume such a special subgoal will not be selected by a computation rule.

We also use a special subgoal, $LOOP$, to mark occurrence of a loop.

Augmenting SLS-trees with the loop cutting and tabling mechanisms leads to the following definition of SLTNF-trees. Here "SLTNF" stands for "*L*inear *T*abulated resolution using a *S*election/computation rule with *N*egation as Finite *F*ailure."

Definition 3.1 (SLTNF-trees) Let P be a logic program, G_0 a top goal, and R a computation rule. Let $\mathcal{T_P}$ be a set of tables each of which contains a finite set of tabled answers. The *SLTNF-tree* $T_{N_0:G_0}$ for $(P \cup \{G_0\}, \mathcal{T_P})$ via R is a tree rooted at $N_0 : G_0$ such that for any node $N_i : G_i$ in the tree with $G_i = \leftarrow L_1, ..., L_n$:

1. If $n = 0$ then N_i is a *success* leaf, marked by $\Box_t$, else if $L_1 = u^*$ then N_i is a *temporarily undefined* leaf, marked by $\Box_{u^*}$, else if $L_1 = LOOP$ then N_i is a *loop* leaf, marked by $\Box_{loop}$.
2. If $L_j = p(.)$ is a positive literal selected by R, we consider two cases:
   (a) If $TB_{L_j} \in \mathcal{T_P}$ with $TB_{L_j} \rightarrow comp = 1$, then for each tabled answer A in $TB_{L_j} \rightarrow ans$, N_i has a child node $N_k : G_k$ where G_k is the resolvent of A and G_i on L_j. In case that $TB_{L_j} \rightarrow ans = \emptyset$, N_i is a *failure* leaf, marked by $\Box_f$.
   (b) Otherwise, for each tabled answer A in $TB_{L_j} \rightarrow ans$ N_i has a child node $N_k : G_k$ where G_k is the resolvent of A and G_i on L_j, and
      i. If N_i is a descendant loop node then it has a child node $N_l :\leftarrow LOOP$.
      ii. Otherwise, for each clause C in P whose head is unifiable with L_j N_i has a child node $N_l : G_l$ where G_l is the resolvent of C and G_i on L_j. If there are neither tabled answers nor clauses applicable to N_i then N_i is a *failure* leaf, marked by $\Box_f$.
3. Let $L_j = \neg A$ be a negative literal selected by R. If A is not ground then N_i is a *flounder* leaf, marked by $\Box_{fl}$, else we consider the following cases:
   (a) If $TB_A \in \mathcal{T_P}$ with $TB_A \rightarrow comp = 1$ and $TB_A \rightarrow ans = \emptyset$, then N_i has only one child node $N_k : G_k$ with $G_k = \leftarrow L_1, ..., L_{j-1}, L_{j+1}, ..., L_n$.
   (b) If $TB_A \in \mathcal{T_P}$ with $TB_A \rightarrow comp = 1$ and $TB_A \rightarrow ans = \{A\}$, then N_i is a *failure* leaf, marked by $\Box_f$.
   (c) Otherwise, if the current SLTNF-tree or one of its ancestor SLTNF-trees is with a goal $\leftarrow A$ at the root, N_i has only one child node $N_k : G_k$ where if $L_n \neq u^*$ then $G_k = \leftarrow L_1, ..., L_{j-1}, L_{j+1}, ..., L_n, u^*$ else $G_k = \leftarrow L_1, ..., L_{j-1}, L_{j+1}, ..., L_n$.
   (d) Otherwise, let $T_{N_r:\leftarrow A}$ be an (subsidiary) SLTNF-tree for $(P \cup \{\leftarrow A\}, \mathcal{T_P})$ via R. We have the following cases:

i. If $T_{N_r:\leftarrow A}$ has a success leaf then N_i is a *failure* leaf, marked by $\Box_f$.

ii. If $T_{N_r:\leftarrow A}$ has no success leaf but a flounder leaf then N_i is a *flounder* leaf, marked by $\Box_{fl}$.

iii. (*Negation As Finite Failure*) If all branches of $T_{N_r:\leftarrow A}$ end at either a failure or a loop leaf where for each loop leaf generated from a descendant loop subgoal V, no successful sub-derivation for its ancestor loop subgoal has a correct answer substitution θ such that $V\theta$ is not in $\mathcal{T_P}$, then N_i has only one child node $N_k : G_k$ with $G_k = \leftarrow L_1, ..., L_{j-1}, L_{j+1}, ..., L_n$.

iv. Otherwise, N_i has only one child node $N_k : G_k$ where if $L_n \neq u^*$ $G_k = \leftarrow L_1, ..., L_{j-1}, L_{j+1}, ..., L_n, u^*$ else $G_k = \leftarrow L_1, ..., L_{j-1}, L_{j+1}, ..., L_n$.

Note that some commonly used concepts, such as derivations (for goals), sub-derivations (for subgoals), sub-trees (for subgoals), generalized trees, and correct answer substitutions, have the same meanings as in SLS-trees (see Section 2).

Positive loops are broken simply by disallowing descendant loop nodes to apply clauses in P for expansion (see case 2b), while negative loops are broken by substituting u^* for looping negative subgoals (see case 3c). This guarantees that SLTNF-trees are finite for logic programs with the bounded-term-size property (see Definition 3.2 and Theorem 3.1).

Note that u^* is only introduced to signify existence of subgoals whose truth values are temporarily non-determined because of occurrence of positive or negative loops. So keeping only one u^* in a goal is enough for such a purpose. From case 1 of Definition 3.1 we see that goals with u^* cannot lead to a success leaf. However, u^* may well appear in a failure leaf since one of the other subgoals may fail regardless of what truth values the temporarily undefined subgoals would take. This achieves the effect of what a preferential computation rule [21] is supposed to achieve, although our computation rule is not necessarily preferential.

Observe that SLTNF-trees implement an *Negation As Finite Failure* (NAF) rule (see case 3(d)iii): A ground subgoal $\neg A$ fails if A succeeds, and succeeds if A finitely fails after exhausting all answers of the loop subgoals involved in evaluating A. This NAF rule is the same as that used in SLDNF-resolution [9] except that loop leaves are considered.

The following example illustrates the process of constructing SLTNF-trees.

Example 3.1 Consider the following program and let $G_0 = \leftarrow p(a, Y)$ be the top goal.

P_1: $p(X, Y) \leftarrow p(X, Z), e(Z, Y).$ $\qquad C_{p_1}$

$p(X, Y) \leftarrow e(X, Y), \neg r.$ $\qquad C_{p_2}$

$e(a, b).$ $\qquad C_{e_1}$

$e(b, c)$ $\qquad C_{e_2}$

$r \leftarrow s, r.$ $\qquad C_r$

$s \leftarrow \neg s.$ $\qquad C_s$

Let $\mathcal{T}_{\mathcal{P}1} = \emptyset$, and for convenience, let us choose the widely-used left-most computation rule (i.e. we always select the left-most subgoal from a goal). The generalized SLTNF-tree $GT_{\leftarrow p(a,Y)}$ for $(P_1 \cup \{\leftarrow p(a,Y)\}, \emptyset)$ is shown in Figure 1,[2] which consists of three finite SLTNF-trees that are rooted at N_0, N_5 and N_8, respectively. Note that two positive loops are cut at N_1 and N_{11}, respectively, and one negative loop is cut at N_9.

$T_{N_5:\leftarrow r}$ has only one branch, which ends at a loop leaf N_{12}. There is no successful sub-derivation for the ancestor loop subgoal r at N_5, so the NAF rule is applicable. Thus, $\neg r$ at N_4 succeeds, leading to a successful sub-derivation for $p(a,Y)$ at N_0 with a correct answer substitution $\{Y/b\}$.

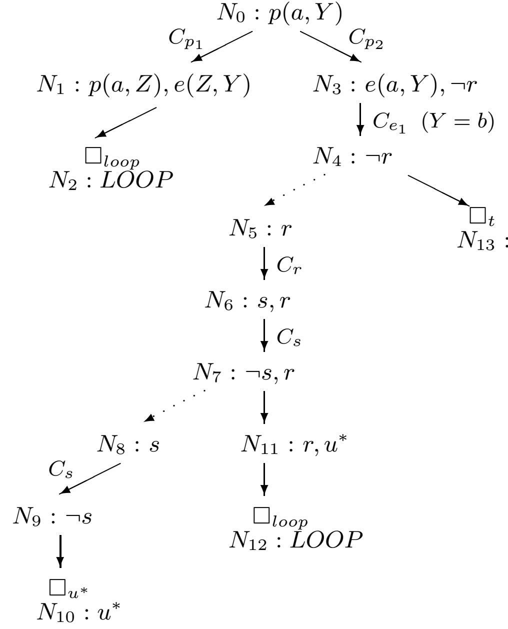

Figure 1: The generalized SLTNF-tree $GT_{\leftarrow p(a,Y)}$ for $(P_1 \cup \{\leftarrow p(a,Y)\}, \emptyset)$.

Definition 3.2 ([30]) A logic program has the *bounded-term-size* property if there is a function $f(n)$ such that whenever a top goal G_0 has no argument whose term size exceeds n, then no subgoals and tabled answers in any generalized SLTNF-tree GT_{G_0} have an argument whose term size exceeds $f(n)$.

The following result shows that the construction of SLTNF-trees is always terminating for logic programs with the bounded-term-size property.

Theorem 3.1 *Let P be a logic program with the bounded-term-size property, G_0 a top goal and R a computation rule. The generalized SLTNF-tree GT_{G_0} for $(P \cup \{G_0\}, \mathcal{T}_{\mathcal{P}})$ via R is finite.*

[2]For simplicity, in depicting SLTNF-trees we omit the "←" symbol in goals.

Proof: First note that GT_{G_0} contains no negative loops (see case 3c). The bounded-term-size property guarantees that no term occurring on any path of GT_{G_0} can have size greater than $f(n)$, where n is a bound on the size of terms in the top goal G_0. Assume, on the contrary, that GT_{G_0} is infinite. Since the branching factor of GT_{G_0} (i.e. the average number of children of all nodes in the tree) is bounded by the finite number of clauses in P, GT_{G_0} either contains an infinite number of SLTNF-trees or has an infinite derivation within some SLTNF-tree. Note that P has only a finite number of predicate, function and constant symbols. If GT_{G_0} contains an infinite number of SLTNF-trees, there must exist negative loops in GT_{G_0}, a contradiction. If GT_{G_0} has an infinite derivation within some SLTNF-tree, some positive subgoal A_0 selected by R must have infinitely many variant descendants $A_1, A_2, ..., A_i, ...$ on the path such that the proof of A_0 needs the proof of A_1 that needs the proof of A_2, and so on. That is, A_i is an ancestor loop subgoal of A_j for any $0 \leq i < j$. This contradicts the fact that any descendant loop subgoal in GT_{G_0} has only one ancestor loop subgoal because a descendant loop subgoal cannot generate descendant loop subgoals since no clauses will be applied to it for expansion (see case 2b of Definition 3.1). $\square$

Consider Figure 1 again. Observe that if we continued expanding N_1 (like Global SLS-resolution) by applying C_{p_1} and C_{p_2}, we would generate another correct answer substitution $\{Y/c\}$ for G_0. This indicates that applying loop cutting alone would result in incompleteness.

We use *answer iteration* [25] to derive all answers of loop subgoals. Here is the basic idea: We first build a generalized SLTNF-tree for $(P \cup \{G_0\}, \mathcal{T_P}^0)$ with $\mathcal{T_P}^0 = \emptyset$ while collecting all new tabled answers (for all tabled subgoals) into NEW^0. Then we build a new generalized SLTNF-tree for $(P \cup \{G_0\}, \mathcal{T_P}^1)$ with $\mathcal{T_P}^1 = \mathcal{T_P}^0 \cup NEW^0$ while collecting all new tabled answers into NEW^1. Such an iterative process continues until no new tabled answers are available.

The key issue with answer iteration is *answer completion*, i.e, how to determine if the table of a subgoal is complete at some derivation stages. Careful reader may have noticed that we have already used a completion criterion for ground subgoals in defining the NAF rule (see case 3d of Definition 3.1). We now generalize this criterion to all subgoals.

Theorem 3.2 *Let GT_{G_0} be the generalized SLTNF-tree for $(P \cup \{G_0\}, \mathcal{T_P})$ and NEW contain all new tabled answers in GT_{G_0}. The following completion criteria hold.*

1. *For a ground tabled positive subgoal A, $TB_A \in \mathcal{T_P} \cup NEW$ is complete for A if $TB_A \rightarrow ans = \{A\}$.*

2. *For any tabled positive subgoal A, $TB_A \in \mathcal{T_P} \cup NEW$ is complete for A if there is a node $N_i : G_i$ in GT_{G_0}, where A is the selected subgoal in G_i and let T_A be the sub-SLTNF-tree for A at N_i, such that (1) T_A has no temporarily undefined leaf, and (2) for each loop leaf in T_A generated from a descendant loop subgoal V, its ancestor loop*

subgoal has no successful sub-derivation with a correct answer substitution θ such that $V\theta$ is not in $TB_V \in \mathcal{T}_\mathcal{P}$.

Proof: The first criterion is straightforward since A is ground. We now prove the second. Note that there are only two cases in which a tabled subgoal A may get new answers via iteration. The first is due to that some temporarily undefined subgoals in the current round would become successful or failed in the future rounds of iteration. In this case, there must be a sub-SLTNF-tree T_A in GT_{G_0} for A with at least one temporarily undefined leaf. Apparently, this case is excluded by condition (1). The second case is due to that some loop subgoals in the current round would produce new answers in the future rounds of iteration. Such new answers are generated in an iterative way, i.e., in the current round descendant loop subgoals consume only existing tabled answers in $\mathcal{T}_\mathcal{P}$ and help generate new answers (which are not in $\mathcal{T}_\mathcal{P}$) for their ancestor loop subgoals. These new answers are then tabled for the descendant loop subgoals to consume in the next round. In this case, there must be a sub-SLTNF-tree T_A in GT_{G_0} for A which contains at least one descendant loop subgoal V such that its ancestor loop subgoal V' has a successful sub-derivation in T_A with a new correct answer substitution not included in $\mathcal{T}_\mathcal{P}$ (this new answer is not consumed by V in the current round but will be consumed in the next round). Obviously, this case is excluded by condition (2). As a result, conditions (1) and (2) together imply that further iteration would generate no new answers for A. Therefore, TB_A is complete for A after merging $\mathcal{T}_\mathcal{P}$ with the new tabled answers in GT_{G_0}. $\Box$

Example 3.2 Consider Figure 1. We cannot apply Theorem 3.2 to determine the completeness of $TB_{p(a,Y)}$ since the ancestor loop subgoal $p(a, Y)$ at N_0 has a successful sub-derivation with an answer $p(a, b)$ not in $\mathcal{T}_{\mathcal{P}1}$. As we can see, applying this new answer to the descendant loop subgoal at N_1 would generate another new answer $p(a, c)$. The completeness of TB_s is not determinable either, since both the two sub-SLTNF-trees for s (rooted at N_6 and N_8, respectively) contain a temporarily undefined leaf. However, by Theorem 3.2, TB_r is complete.

Definition 3.3 (SLTNF-resolution) Let P be a logic program, $G_0 = \leftarrow A$ a top goal with A an atom, and R a computation rule. Let $\mathcal{T}_\mathcal{P}{}^0 = \emptyset$. *SLTNF-resolution* evaluates G_0 by calling the function $SLTNF(P, G_0, R, \mathcal{T}_\mathcal{P}{}^0)$, defined as follows.

function $SLTNF(P, G_0, R, \mathcal{T}_\mathcal{P}{}^i)$ **returns** a table TB_A
{
  Build a generalized SLTNF-tree $GT^i_{G_0}$ for $(P \cup \{G_0\}, \mathcal{T}_\mathcal{P}{}^i)$ while collecting
    all new tabled answers into NEW^i;
  $\mathcal{T}_\mathcal{P}{}^{i+1} = \mathcal{T}_\mathcal{P}{}^i \cup NEW^i$;
  Check completeness of all tables in $\mathcal{T}_\mathcal{P}{}^{i+1}$ and update their status;

if $NEW^i = \emptyset$ or $TB_A \rightarrow comp = 1$ **then return** TB_A;
return $SLTNF(P, G_0, R, \mathcal{T_P}^{i+1})$;
}

Example 3.3 (Cont. of Example 3.1) First execute $SLTNF(P_1, G_0, R, \mathcal{T_P}_1^0)$ where $\mathcal{T_P}_1^0 = \emptyset$, $G_0 = \leftarrow p(a, Y)$ and R is the left-most computation rule. The procedure builds a generalized SLTNF-tree for $(P_1 \cup \{\leftarrow p(a, Y)\}, \emptyset)$ as shown in Figure 1. It also collects the following new tabled answer into NEW^0: $p(a, b)$ for $TB_{p(a,Y)}$. Moreover, it has TB_r completed by setting $TB_r \rightarrow comp$ to 1 (note that $TB_r \rightarrow ans = \emptyset$).

Next execute $SLTNF(P_1, G_0, R, \mathcal{T_P}_1^1)$ where $\mathcal{T_P}_1^1 = \mathcal{T_P}_1^0 \cup NEW^0$. It builds a generalized SLTNF-tree $GT^1_{\leftarrow p(a,Y)}$ for $(P_1 \cup \{\leftarrow p(a, Y)\}, \mathcal{T_P}_1^1)$ as shown in Figure 2, and collects the following new tabled answer into NEW^1: $p(a, c)$ for $TB_{p(a,Y)}$.

Finally execute $SLTNF(P_1, G_0, R, \mathcal{T_P}_1^2)$ where $\mathcal{T_P}_1^2 = \mathcal{T_P}_1^1 \cup NEW^1$. The procedure builds a generalized SLTNF-tree $GT^2_{\leftarrow p(a,Y)}$ for $(P_1 \cup \{\leftarrow p(a, Y)\}, \mathcal{T_P}_1^2)$ in which no new tabled answer is produced. Therefore, it returns with two tabled answers, $p(a, b)$ and $p(a, c)$, to the top goal G_0.

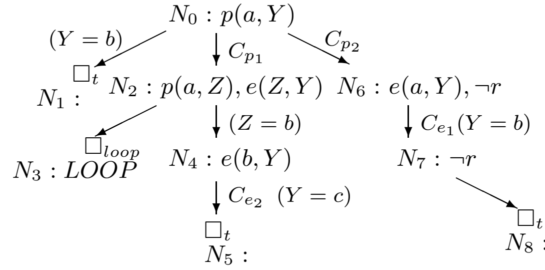

Figure 2: The generalized SLTNF-tree $GT^1_{\leftarrow p(a,Y)}$ for $(P_1 \cup \{\leftarrow p(a, Y)\}, \mathcal{T_P}_1^1)$.

Theorem 3.3 *Let P be a logic program with the bounded-term-size property, G_0 a top goal and R a computation rule. $SLTNF(P, G_0, R, \emptyset)$ terminates in finite time.*

Proof: Let n be the maximum size of arguments in any top goal. Since P has the bounded-term-size property, neither subgoals nor tabled answers have arguments whose size exceeds $f(n)$ for some function f. Let $s = f(n)$. Since P has a finite number of predicate symbols, the number of distinct subgoals (up to variable renaming) occurring in all $GT^i_{G_0}$s is bounded by a finite number $N(s)$. Therefore, SLTNF-resolution performs at most $N(s)$ iterations (i.e. generates at most $N(s)$ generalized SLTNF-trees). By Theorem 3.1, each iteration terminates in finite time, hence SLTNF-resolution terminates in finite time. $\Box$

Theorem 3.4 *Let P be a logic program with the bounded-term-size property, A an atom, and $G_0 = \leftarrow A$ a top goal with A a non-floundering query. Let TB_A be the tabled answers returned from $SLTNF(P, G_0, R, \emptyset)$, and let $T_{N_0:G_0}$ be the SLS-tree for $P \cup \{G_0\}$ via R.*

1. *$A\theta$ is in TB_A if and only if there is a correct answer substitution θ for G_0 in $T_{N_0:G_0}$.*

2. *$TB_A \rightarrow comp = 1$ and $TB_A \rightarrow ans = \emptyset$ if and only if $T_{N_0:G_0}$ is failed.*

Proof: We first prove that SLS-trees with negative loops can be transformed into equivalent SLS-trees without negative loops. Let $T_{N_i:\leftarrow B}$ be an SLS-tree with a descendant SLS-tree $T_{N_j:\leftarrow B}$. Obviously, this is a negative loop. Observe that B at N_i being successful or failed must be independent of the loop SLS-tree $T_{N_j:\leftarrow B}$, for otherwise the truth value of B would depend on $\neg B$ so that B is undefined. This strongly suggests that using a temporarily undefined value u^* as the truth value of $T_{N_j:\leftarrow B}$ does not change the answer of B at N_i. In other words, any SLS-trees with negative loops can be transformed into equivalent SLS-trees where all descendant loop SLS-trees are assumed to return a temporarily undefined value u^*.

Let $T^i_{N_0:G_0}$ and $GT^i_{G_0}$ be respectively the SLTNF-tree and the generalized SLTNF-tree for $(P \cup \{G_0\}, \mathcal{T_P}^i)$, where $\mathcal{T_P}^0 = \emptyset$ and for each $i \geq 0$, $\mathcal{T_P}^{i+1} = \mathcal{T_P}^i \cup NEW^i$ where NEW^i contains all new tabled answers collected from $GT^i_{G_0}$. We prove this theorem by showing that answers over SLS-derivations can be extracted in an iterative way and such iterations are the same as those of SLTNF-resolution. Therefore, both resolutions extract the same set of answers to G_0. We distinguish between three cases:

1. For any answer $A\theta$ that is generated without going through any loops, we must have the same successful derivations for A in $T^0_{N_0:G_0}$ as in $T_{N_0:G_0}$.

2. Let us consider answers to G_0 that are generated without going through any negative loops. Without loss of generality, assume the SLS-derivations for the answers involve positive loops as shown in Figure 3, where for any $j > k \geq 0$, B^k is an ancestor loop subgoal of B^j and each T^k together with the branch leading to $N_{i^{k+1}}$ is a sub-SLS-tree for B^k at N_{i^k}. Obviously, all T^ks are identical up to variable renaming and thus they have the same set S_{B^0} of correct answer substitutions for B^k (up to variable renaming).

   Observe that besides S_{B^0}, the other possible correct answer substitutions for B^k must be generated via the infinite loops in an iterative way: For any $l > 0$, the correct answer substitutions for $B^l, E^l_1, ..., E^l_n$ at N_{i^l} combined with δ^l, when restricted to the variables in B^{l-1}, are also correct answer substitutions for B^{l-1} at $N_{i^{l-1}}$. These substitutions are obtained by applying each correct answer substitution θ^l for B^l to $E^l_1, ..., E^l_n$ and then evaluating $(E^l_1, ..., E^l_n)\theta^l$. Since P has the bounded-term-size property, no correct answer substitution requires performing an infinite number of such iterations. That is, there must exist a depth bound d such that any correct answer substitution θ for B^0 is in the following closure (fixpoint):

   - The initial set of correct answer substitutions is $S_d = S_{B^0}$.

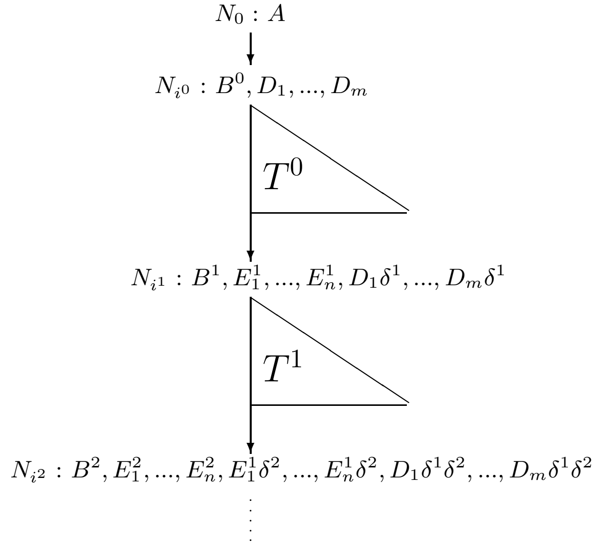

Figure 3: SLS-derivations with positive loops.

- For each $0 < l \leq d$, the set of correct answer substitutions for B^{l-1} at $N_{i^{l-1}}$ is $S_{l-1} = S_l \cup \{\theta | \theta^l \in S_l$ and $\theta = \delta^l \theta^l \alpha$ where α is a correct answer substitution for $(E_1^l, ..., E_n^l)\theta^l\}$.

Apparently, SLTNF-resolution performs the same iterations by making use of the loop cutting and tabling mechanisms: In the beginning, TB_{B^0} is empty. The loop is cut at N_{i^1}, so $TB_{B^0} = S_d = S_{B^0}$ after $T^0_{N_0:G_0}$ is generated (note B^0 and B^k (resp., T^0 and T^k) are variants). Then for the l-th iteration $(0 < l \leq d)$ TB_{B^0} obtains new answers by applying the already tabled answers to B^1 at N_{i^1} in $T^l_{N_0:G_0}$; i.e., $TB_{B^0} = S_{l-1}$. As a result, SLS-resolution and SLTNF-resolution derive the same set of correct answer substitutions for all subgoals involving no negative loops.

3. Let us now consider answers to G_0 that are generated involving negative loops. As we discussed earlier, loop descendant SLS-trees $T_{N_i:\leftarrow B}$ can be removed by assuming they return a temporarily undefined value u^*. Then we get equivalent SLS-trees without any negative loops. By point 2 above, we can exhaust all answers to G_0 from these (negative loop free) SLS-trees in an iterative way, as SLTNF-resolution does. If no single answer to A in G_0 is generated after the iteration, we have two cases. The first is that no SLS-derivation for A at N_0 ends at a leaf with u^*. This means that the truth value of A does not depend on any negative loop subgoal, so $T_{N_0:G_0}$ is failed and thus $TB_A \rightarrow comp = 1$ and $TB_A \rightarrow ans = \emptyset$. The second case is that some SLS-derivation for A at N_0 ends at a leaf with u^*. This means that the truth value of A recursively depends on some negative loop subgoal, so A is undefined. In this case, SLTNF-resolution stops with $TB_A \rightarrow comp = 0$ and $TB_A \rightarrow ans = \emptyset$. $\square$

Since Global SLS-resolution is sound and complete w.r.t. the well-founded semantics (see Theorem 2.1), we have the following immediate corollary.

Corollary 3.5 *Let P be a logic program, R a computation rule, and $G_0 \leftarrow Q$ be a top goal with Q a non-floundering query under R. SLTNF-resolution is sound and complete w.r.t. the well-founded semantics.*

4 Related Work

Existing procedural semantics for the well-founded model can be divided into two groups in terms of the way they make derivations: (1) *bottom-up* approaches, such as the alternating fixpoint approach [31, 15], the magic sets approach [12, 16] and the transformation-based bottom-up approach [4, 5, 6], and (2) *top-down* approaches. Our method belongs to the second group. Existing top-down methods can be further divided into two groups: (1) non-tabling methods, such as Global SLS-resolution, and (2) tabling methods. Our method is one with tabling. Several tabling methods for positive logic programs have been proposed, such as OLDT-resolution [29], TP-resolution [25, 36] and the DRA tabling mechanism [11]. However, to the best of our knowledge, only SLG-resolution and SLT-resolution use tabling to compute the well-founded semantics for general logic programs.

SLG-resolution is the state-of-the-art tabling mechanism. It is based on program transformations, instead of on standard tree-based formulations like SLDNF- or Global SLS-resolution. Starting from the predicates of the top goal, it transforms (instantiates) a set of clauses, called a *system*, into another system based on six basic transformation rules. Such a system corresponds to a forest of trees with each tree rooted at a tabled subgoal. A special class of literals, called *delaying literals*, is used to represent and handle temporarily undefined negative literals. Negative loops are identified by maintaining an additional *dependency graph* of subgoals [7, 8]. In contrast, SLTNF-resolution generates an SLTNF-tree for the top goal in which the flow of the query evaluation is naturally depicted by the ordered expansions of tree nodes. Such a tree-style formulation is quite easy for users to understand and keep track of the computation. It can also be implemented efficiently using a simple stack-based memory structure. The disadvantage of SLTNF-resolution is that it is a little more costly in time than SLG-resolution due to the use of answer iteration in exchange for the linearity of derivations.

SLT-resolution is a tabling mechanism with the linearity property. Like SLTNF-resolution, it expands tree nodes by first applying tabled answers and then applying clauses. It also uses answer iteration to derive missing answers caused by loop cuttings. However, it is different from SLTNF-resolution both in loop handling and in answer completion (note that loop handling and answer completion are two key components of a tabling system).

Recall that SLT-resolution defines positive and negative loops based on the same ancestor-descendant relation: Let A be a selected positive subgoal and B be a subgoal produced by applying a clause to A, then B is a descendant subgoal of A and inherits all ancester subgoals of A; let $\neg A$ be a selected ground subgoal with $T_{N_r:\leftarrow A}$ being its subsidiary SLT-tree, then the subgoal A at N_r inherits all ancester subgoals of $\neg A$. A (positive or negative) loop occurs when a selected subgoal has an ancestor loop subgoal. Observe that the ancestor and descendant subgoals may be in different SLT-trees.

When a positive loop occurs, SLTNF-resolution will apply no clauses to the descendant loop subgoal for node expansion, which guarantees that any ancestor loop subgoal has just one descendant loop subgoal. However, SLT-resolution will continue expanding the descendant loop subgoal by applying those clauses that have not yet been applied by any of its ancestor loop subgoals. As an illustration, in Figure 1, SLT-resolution will apply C_{p_2} to expand N_1, leading to a child node N_1' with a goal $\leftarrow e(a,Z), \neg r, e(Z,Y)$. Observe that if the subgoal $e(a,Z)$ at N_1' were $p(a,Z)$, another loop would occur between N_0 and N_1'. This suggests that in SLT-resolution, an ancestor loop subgoal may have several descendant loop subgoals. Due to this, SLT-resolution is more complicated and costly than SLTNF-resolution in handling positive loops.

SLT-resolution is also more costly than SLTNF-resolution in handling negative loops. It checks negative loops in the same way as positive loops by comparing a selected subgoal with all of its ancester subgoals across all of its ancestor SLT-trees. However, in SLTNF-resolution a negative loop is checked simply by comparing a selected ground negative subgoal with the root goals of its ancestor SLTNF-trees. Recall that a negative loop occurs if a negative ground subgoal $\neg A$ is selected such that the root of the current SLTNF-tree or one of its ancestor SLTNF-trees is with a goal $\leftarrow A$.

SLT-resolution provides no mechanism for answer completion except that when a generalized SLT-tree $GT^i_{G_0}$ is generated which contains no new tabled answers, it evaluates each negative ground subgoal $\neg A$ in $GT^i_{G_0}$ in a way such that (1) $\neg A$ fails if A is a tabled answer, and (2) $\neg A$ succeeds if (i) all branches of its subsidiary SLT-tree $T_{N_r:\leftarrow A}$ end with a failure leaf and (ii) for each loop subgoal in $T_{N_r:\leftarrow A}$, all branches of the sub-SLT-trees for its ancestor loop subgoals end with a failure leaf. Not only is this process complicated, it is also quite inefficient since the evaluation of $\neg A$ may involve several ancestor SLT-trees. In contrast, SLTNF-resolution provides two criteria for completing answers of both negative and positive subgoals. On the one hand, the criteria are applied during the construction of generalized SLT-trees so that redundant derivations can be pruned as early as possible. On the other hand, checking the completion of a subgoal involves only one SLTNF-tree.

5 Conclusions and Further Work

Global SLS-resolution and SLG-resolution represent two typical styles in top-down computing the well-founded semantics; the former emphasizes the linearity of derivations as SLDNF-resolution does while the latter focuses on making full use of tabling to resolve loops and redundant computations. SLTNF-resolution obtains the advantages of the two methods by enhancing Global SLS-resolution with loop cutting and tabling mechanisms. It seems that the existing linear tabling mechanism SLT-resolution has similar advantages, but SLTNF-resolution is simpler and more efficient due to its distinct mechanisms for loop handling and answer completion.

Due to its SLDNF-tree like structure, SLTNF-resolution can be implemented over a Prolog abstract machine such as WAM [33] or ATOAM [35]. In particular, it can be implemented over existing linear tabling systems for positive logic programs such as [36, 37, 38], simply by adding two more mechanisms, one for identifying negative loops and the other for checking answer completion of tabled subgoals. We are currently working on the implementation. Experimental analysis of SLTNF-resolution will then be reported in the near future.

Acknowledgment

We thank the anonymous referees for their helpful comments. Yi-Dong Shen 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] K. R . Apt, H. A. Blair and A. Walker, Towards a theory of declarative knowledge. In: (J. Minker ed.) *Foundations of Deductive Databases and Logic Programming*, Morgan Kaufmann, 1988, pp. 89-148.

[2] R. N. Bol, K. R. Apt and J. W. Klop, An analysis of loop checking mechanisms for logic programs, *Theoretical Computer Science* 86(1):35-79 (1991).

[3] R. N. Bol and L. Degerstedt, Tabulated resolution for the well-founded semantics. *Journal of Logic Programming* 34(2):67-109 (1998).

[4] S. Brass and J. Dix, Characterizations of the disjunctive well-founded semantics: confluent calculi and iterated GCWA, *Journal of Automated Reasoning* 20(1):143-165 (1998).

[5] S. Brass, J. Dix, B. Freitag and U. Zukowski, Transformation-based bottom-up computation of the well-founded model, *Theory and Practice of Logic Programming* 1 (5):497-538 (2001).

[6] S. Brass, J. Dix, I. Niemelae and T. Przymusinski, On the equivalence of the STATIC and disjunctive Well-founded Semantics and their Computation, *Theoretical Computer Science* 258(1-2):523-553 (2001).

[7] W. D. Chen, T. Swift and D. S. Warren, Efficient top-down computation of queries under the well-founded semantics, *Journal of Logic Programming* 24(3):161-199 (1995).

[8] W. D. Chen and D. S. Warren, Tabled evaluation with delaying for general logic programs, *J. ACM* 43(1):20-74 (1996).

[9] K. L. Clark, Negation as Failure, in: (H. Gallaire and J. Minker, eds.) *Logic and Databases*, Plenum, New York, 1978, pp. 293-322.

[10] D. De Schreye and S. Decorte, Termination of logic programs: the never-ending story, *Journal of Logic Programming* 19/20:199-260 (1993).

[11] H. F. Guo and G. Gupta, A simple scheme for implementing tabled logic programming systems based on dynamic reordering of alternatives, in: (P. Codognet, ed.) *Lecture Notes in Computer Science* 2237, 2001, pp. 181-196.

[12] D. B. Kemp, D. Srivastava and P. J. Stuckey, Bottom-up evaluation and query optimization of well-founded models, *Theoretical Computer Science* 146:145-184 (1995).

[13] R. A. Kowalski, Predicate logic as a programming language, *IFIP* 74, pp.569-574.

[14] J. W. Lloyd, *Foundations of Logic Programming*, 2nd ed., Springer-Verlag, Berlin, 1987.

[15] Z. Lonc and M. Truszczynski, On the problem of computing the well-founded semantics, *Theory and Practice of Logic Programming* 1 (5):591-609 (2001).

[16] S. Morishita, An extension to Van Gelder's alternating fixpoint to magic programs, *Journal of Computer and System Science* 52:506-521 (1996).

[17] T. Przymusinski, 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, pp. 11-21.

[18] H. Przymusinska, T. Przymusinski and H. Seki, Soundness and completeness of partial deductions for well-founded semantics, in: *Proc. of the International Conference on Automated Reasoning*, 1992.

[19] T. Przymusinski, The well-founded semantics coincides with the three-valued stable semantics, *Fundamenta Informaticae* 13:445-463 (1990).

[20] J. A. Robinson, A machine-oriented logic based on the resolution principle, *J. ACM* 12(1):23-41 (1965).

[21] K. Ross, A procedural semantics for well-founded negation in logic programs, *Journal of Logic Programming* 13(1):1-22 (1992).

[22] K. Sagonas, T. Swift and D. S. Warren, XSB as an efficient deductive database engine, in: *Proc. of the ACM SIGMOD Conference on Management of Data*, Minneapolis, 1994, pp. 442-453.

[23] K. Sagonas, T. Swift, D. S. Warren, J. Freire and P. Rao, *The XSB Programmer's Manual (Version 1.8)*, 1998.

[24] Y. D. Shen, J. H. You, L. Y. Yuan, S. P. Shen and Q. Yang, A dynamic approach to characterizing termination of general logic programs, *ACM Transactions on Computational Logic* 4(4):417-430 (2003).

[25] Y. D. Shen, L. Y. Yuan, J. H. You, and N. F. Zhou, Linear tabulated resolution based on Prolog control strategy, *Theory and Practice of Logic Programming* 1 (1):71-103 (2001).

[26] Y. D. Shen, L. Y. Yuan and J. H. You, SLT-resolution for the well-founded semantics, *Journal of Automated Reasoning* 28(1): 53-97 (2002).

[27] J. C. Shepherdson, Negation in logic programming, in: (J. Minker, ed.) *Foundations of Deductive Databases and Logic Programming*, Morgan Kaufmann, 1988, pp. 19-88.

[28] T. Swift, Tabling for non-monotonic programming, *Annals of Mathematics and Artificial Intelligence* 25 (3-4):201-240 (1999).

[29] H. Tamaki and T. Sato, OLD resolution with tabulation, in: *Proc. of the Third International Conference on Logic Programming*, London, 1986, pp. 84-98.

[30] A. Van Gelder, Negation as failure using tight derivations for general logic programs, *Journal of Logic Programming* 6(1&2):109-133 (1989).

[31] A. Van Gelder, The alternating fixpoint of logic programs with negation, in: *Proc. of the 8th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, 1989, pp. 1-10.

[32] A. Van Gelder, K. Ross, J. Schlipf, The well-founded semantics for general logic programs, *J. ACM* 38(3):620-650 (1991).

[33] D. H. D. Warren, An abstract Prolog instruction set, Technical Report 309, SRI International, 1983.

[34] D. S. Warren, Memoing for logic programs, *CACM* 35(3):93-111 (1992).

[35] N. F. Zhou, Parameter passing and control stack management in Prolog implementation revisited, *ACM Transactions on Programming Languages and Systems* 18(6):752-779 (1996).

[36] N. F. Zhou, Y. D. Shen, L. Yuan and J. You, Implementation of linear tabling mechanisms, *Journal of Functional and Logic Programming* (10):1-17 (2001).

[37] N. F. Zhou and T. Sato, Efficient fixpoint computation in linear tabling, in: *The Fifth ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming*, Sweden, 2003.

[38] N. F. Zhou, Y. D. Shen and T. Sato, Semi-naive evaluation in linear tabling, in: *The Sixth ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming*, Verona, Italy, 2004.