# EFFICIENT TOP-DOWN COMPUTATION OF QUERIES UNDER THE WELL-FOUNDED SEMANTICS

**WEIDONG CHEN * , TERRANCE SWIFT† , AND DAVID S. WARREN‡**

▷

The well-founded model provides a natural and robust semantics for logic programs with negative literals in rule bodies. Although various procedural semantics have been proposed for query evaluation under the well-founded semantics, the practical issues of implementation for effective and efficient computation of queries have been rarely discussed.

This paper investigates two major implementation issues of query evaluation under the well-founded semantics, namely (a) to ensure that negative literals be resolved only after their positive counterparts have been completely evaluated, and (b) to detect and handle potential negative loops. We present efficient incremental algorithms for maintaining positive and negative dependencies among subgoals in a top-down evaluation. Both completely evaluated subgoals and potential negative loops are detected by inspecting the dependency information of a *single* subgoal. Our implementation can be viewed as an effective successor to SLDNF resolution, extending Prolog computation in a natural and smooth way.

◁

*Supported in part by the National Science Foundation under Grant No. IRI-9212074.

*Address correspondence to* Weidong Chen, Department of Computer Science and Engineering, Southern Methodist University, Dallas, TX 75275-0122.

†Department of Computer Science, SUNY at Stony Brook, Stony Brook, NY 11794-4400.

‡Department of Computer Science, SUNY at Stony Brook, Stony Brook, NY 11794-4400.

Supported in part by the National Science Foundation under Grant No. CCR-9102159.

*THE JOURNAL OF LOGIC PROGRAMMING*


655 Avenue of the Americas, New York, NY 10010 0743-1066/93/$3.50

## 1. INTRODUCTION

The well-founded semantics [29] provides a natural and robust declarative meaning to all logic programs with negation in rule bodies. Practical use of the well-founded semantics, however, depends upon the implementation of an effective and efficient query evaluation procedure. Although various procedural semantics have been proposed, implementation techniques for the well-founded semantics have not yet received adequate attention.

Earlier procedures for the well-founded semantics by Przymusinski [16] and Ross [20] are extensions of SLDNF resolution with infinite failure. They are not suitable for effective computation of queries due to possible infinite loops even when programs are function-free.

Effective top-down computation with tabling is explored in Well! [2] and XOLDTNF [5] for the well-founded semantics. Two aspects of these approaches should be noted. First, a ground negative subgoal is solved by computing its positive counterpart up to a fixpoint as in Prolog. The fixpoint computation is a simple mechanism to guarantee that the positive counterpart of a negative literal be completely evaluated. Second, to prevent negative loops, each subgoal has an associated set of ground negative literals, called a *negative context*. When a ground negative literal is selected, there is a negative loop if it is already in the negative context of the current subgoal. These two mechanisms, however, prohibit the full sharing of answers to subgoals across different negative contexts in the nested fixpoint computation. Although simple to implement, they may cause exponential behavior in the worst case [7].

Bottom-up computation of the well-founded semantics has also been studied [10, 11, 13, 15]. These approaches are based upon either van Gelder's alternating fixpoint characterization of the well-founded model [28] or the fixpoint for the smallest three valued stable model [4, 17]. Due to the single fixpoint computation, all answers of subgoals can be shared. Each iteration of the fixpoint computation, however, may over-estimate the truth or undefinedness of negative subgoals. This over-estimate is necessary for non-stratified programs in general, but should be properly controlled so as to avoid evaluating irrelevant subgoals.

The first work on controlling the search in bottom-up computation is reported in [18] for left-to-right modularly stratified programs. The Ordered Search technique in [18] attempts to capture relevant subgoals in a top-down fashion by controlling the availability of magic tuples (that represent calls in a top-down computation). This is achieved by maintaining subgoal dependencies in a sequence of so called *ContextNodes*. The idea of subgoal dependencies can be traced back to [21], where they were used to determine if subgoals were completely evaluated. However, the issue of efficient dependency maintenance was not investigated in detail.

Our work on effective computation of the well-founded semantics started with XOLDTNF [5]. As we have mentioned, XOLDTNF uses a fixpoint computation to guarantee that the positive counterpart of a negative literal be completely evaluated, and it uses negative contexts for handling negative loops. Both mechanisms may cause redundant computation. To resolve this problem, we investigated the idea of subgoal dependencies, which proved to provide a simple solution to both completion of subgoals and detection of negative loops. The conceptual framework of this new approach, called *SLG resolution*, was reported in [6]. It is goal-oriented and has a polynomial data complexity for function-free programs. Detailed proofs

can be found in [7] for the soundness and search space completeness of SLG resolution with respect to three valued stable models, including the well founded partial model as a special case. A similar framework is presented in [3]. A meta interpreter implementation integrating Prolog and SLG resolution, called *The SLG System* [8], is available by anonymous FTP from seas.smu.edu or cs.sunysb.edu. A WAM-based compiler implementation integrating Prolog and the restricted SLG resolution for left-to-right modularly stratified programs, called *The XSB Logic Programming System*, [22], has been released and is available by anonymous FTP from cs.sunysb.edu.

As a conceptual framework, SLG resolution consists of a number of transformations by which a query is reduced to a set of answer clauses, but it does not specify *in what order* these transformations should be applied. Two important transformations are COMPLETION and DELAYING. COMPLETION detects subgoals that have been completely evaluated so that their negative counterparts can be resolved. DELAYING delays ground negative literals so that computation can proceed even in case of negative loops. Delaying in SLG resolution corresponds to over-estimating the truth or undefinedness of negative subgoals in bottom-up computation. To avoid computation of irrelevant subgoals, delaying should be tightly controlled.

This paper addresses the fundamental issues of implementation that are common in both top-down and bottom-up computation of the well-founded semantics. In particular, we present incremental algorithms for maintaining dependencies among subgoals. By inspecting the dependency information of a single subgoal, we can determine efficiently if subgoals are completely evaluated or are possibly involved in negative loops.

Practically, the XSB system implementing SLG resolution for left-to-right modularly stratified programs is upwardly compatible with Prolog. With a few simple declarations for the XSB compiler, either given by the user or generated by the system, Prolog programs can be executed using SLG resolution, SLDNF resolution, or a mixture of the two. At an operational level, implementing SLG in a WAM-based framework not only allows for smooth integration of the deductive database and logic programming paradigms; it also allows the SLG engine to benefit from the highly optimized unification and control algorithms in the WAM. As a simple instance, a left linear ancestor predicate executed using SLG spends 70% of the time on WAM instructions, and about 30% of the time on instructions created for SLG.

The rest of the paper is organized as follows. Section 2 describes the search forest and the corresponding subgoal graph that may be induced by transformations of SLG resolution in query evaluation. Section 3 introduces the main issues in incremental maintenance of subgoal dependencies during query evaluation. Section 4 presents detailed algorithms in our implementation of SLG resolution. Section 5 concludes with a summary and a comparison with related work.

## 2. SLG RESOLUTION FOR WELL-FOUNDED SEMANTICS

This section reviews briefly the well-founded semantics of logic programs [29] and discusses the search forest and dependency graph for query evaluation. The basic framework of SLG resolution and its correctness theorem [6, 7] are described, which will be used to establish the correctness of our implementation.

### 2.1. Well-Founded Semantics

We assume the basic terminology of logic programs [12]. A *program* is a finite set of clauses of the form:

$$A \text{ :- } L_1, ..., L_n$$

where $A$ is an atom and $L_1, ..., L_n$ are literals. When $n = 0$, a clause, possibly containing variables, is called a *fact*. By a *subgoal* we mean an atom. Subgoals (and literals) that are variants of each other are considered syntactically identical.

The *Herbrand universe* of a program $P$ is the set of all ground terms that may be constructed from the constants and function symbols appearing in $P$. An arbitrary constant is added if no constant occurs in $P$. The *Herbrand base* of $P$, denoted by $\mathcal{B}_P$, is the set of all ground atoms with predicates occurring in $P$ whose arguments are in the Herbrand universe of $P$. The *Herbrand instantiation* of $P$ is the (possibly infinite) set of all ground clauses obtained by substituting terms in the Herbrand universe for variables in clauses in $P$.

Let $P$ be a logic program and $\mathcal{B}_P$ be the Herbrand base of $P$. A set $I$ of ground literals is *consistent* if for no ground atom $A$, both $A$ and $\sim A$ are in $I$. An *interpretation* $I$ is a consistent set of ground literals.

The well-founded semantics depends upon the notion of *unfounded sets* to derive atoms that are false.

*Definition 2.1.* [29] Let $P$ be a logic program, $I$ be an interpretation, and $U$ be a subset of the Herbrand base $\mathcal{B}_P$. $U$ is an *unfounded set of $P$ with respect to $I$* if every atom $A \in U$ satisfies the following condition: for every ground instance of a clause in $P$ whose head is $A$, either

- some literal $L$ in the body is false in $I$; or
- some positive literal $L$ in the body is also in $U$.

The union of all unfounded sets of $P$ with respect to $I$ coincides with the *greatest unfounded set of $P$ with respect to $I$*, denoted by $\mathbf{U}_P(I)$.

Intuitively if a set of atoms depends upon each other through positive literals and there is no escape clause for any of the atoms, then the set is unfounded and all atoms in the set will be false in the well-founded semantics.

*Definition 2.2.* [29] Let $P$ be a logic program, and $I$ be an interpretation. Transformations $\mathbf{T}_P$ and $\mathbf{W}_P$ are defined as follows:

- $A \in \mathbf{T}_P(I)$ if and only if there is a ground instance of some clause in $P$ with head $A$ such that all literals in the body are true in $I$;
- $\mathbf{W}_P(I) = \mathbf{T}_P(I) \cup \{\sim A | A \in \mathbf{U}_P(I)\}$.

Transformations $\mathbf{T}_P$ and $\mathbf{W}_P$ are known to be monotonic [29]. The powers $\mathbf{W}_P^\alpha$ are defined in the standard manner, where $\alpha$ ranges over all countable ordinals. The well-founded partial model of a program $P$, denoted by $WF(P)$, is the union of all $\mathbf{W}_P^\alpha$.

### 2.2. Search Forest and Dependency Graph

In SLG resolution [6], query evaluation is viewed as traversing a search tree or a search forest for a query. This subsection describes the search forest and the corresponding dependency graph of subgoals for a query.

*2.2.1. SLD Resolution with Tabling* For programs without negation, SLG resolution reduces to SLD resolution with tabling [9, 27, 30]. In all the examples, we use a left-most computation rule although an arbitrary but fixed computation rule is allowed.

Let $P$ be a program without negation and $A$ be a subgoal. We construct a *search forest* for $A$ with respect to $P$. Each node in the forest is labeled by a clause. Initially, the search forest has one tree, namely the tree for $A$, whose root node is labeled by $A$ :- $A$.

The root node of the tree for a subgoal $A$, labeled by $A$ :- $A$, has a child node for each resolvent of $A$ :- $A$ with a clause in $P$ on the $A$ in the body of $A$ :- $A$.

If a node is labeled by a fact $B$ in the tree for a subgoal $A$, then $B$ is an answer for $A$. Two answers are considered identical if they are renaming variants of each other.

Let $v$ be a non-root node in the tree for subgoal $A$, $G$ be the clause labeling $v$, and $B$ be the selected atom of $G$. If the current search forest does not contain the tree for subgoal $B$, the tree for $B$ is added, whose root node is labeled by $B$ :- $B$. For each (distinct) answer $B'$ of $B$, $v$ has a child that is labeled by the resolvent of $G$ with $B'$ on the selected atom $B$. This process continues until no new node or new tree can be created.

*Example 2.1.* [5] Consider a small cyclic graph and the common definition of transitive closure:

$e(a,b).\ e(b,c).\ e(b,a).$
$tc(X,Y)$ :- $e(X,Y).$
$tc(X,Y)$ :- $e(X,Z), tc(Z,Y).$

Figure 1 shows the search forest for subgoal $tc(a,V)$. (Trees for subgoals of predicate $e/2$ are not shown.)

Corresponding to each search forest, there is a dependency graph of subgoals. Each node in the dependency graph is a subgoal. An edge from a subgoal $A$ to a subgoal $B$ corresponds to a non-root node $v$ in the tree for $A$ such that $B$ is the subgoal of the selected literal from the label of $v$.

For instance, the tree for $tc(a,V)$ contains a non-root node labeled by $tc(a,V)$ :- $tc(b,V)$. It determines an edge in the dependency graph from $tc(a,V)$ to $tc(b,V)$, the selected atom of the label of the non-root node. The dependency graph corresponding to the forest in Figure 1 is shown in Figure 2. The intuition behind the dependency graph is that it contains a path from subgoal $A$ to subgoal $B$ if the truth value of $A$ may depend in some way on the truth value of $B$.

*2.2.2. Stratified Negation* For stratified programs [1], one issue is how to ensure that a ground subgoal be completely evaluated so that the success of its negative counterpart can be determined. A negative literal can succeed only if the corresponding positive subgoal has no answers after having been completely evaluated.

The notion of a search forest can be extended to stratified programs in a straightforward way. When a ground negative literal $\sim B$ is selected, we start the tree for $B$ if the current search forest does not contain the tree for $B$. If $B$ succeeds with an answer, then every node with $\sim B$ selected is marked as failed. If $B$ is completely

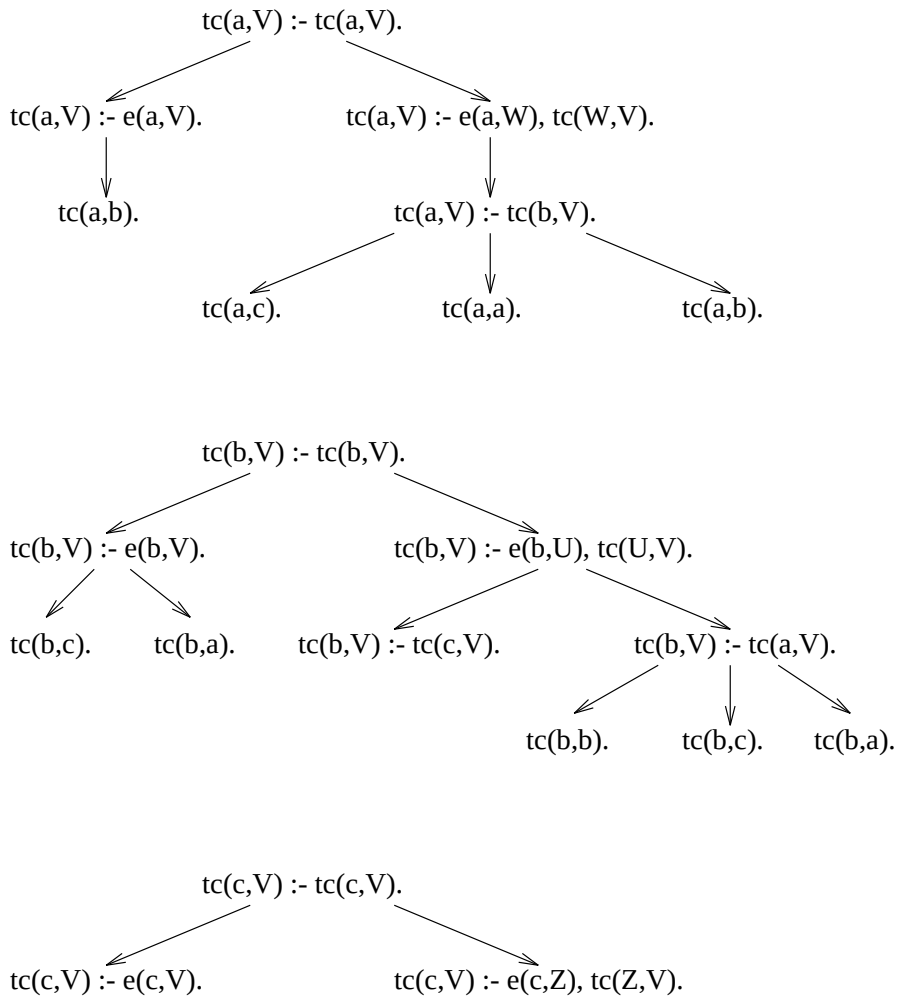


**Figure 1.** Search forest for $tc(a, V)$

evaluated and has no answers, then $\sim B$ succeeds and every node with $\sim B$ selected has a single child node obtained by deleting $\sim B$.

*Example 2.2.* Consider the following program and subgoal $m(c)$:

$m(X)$ :- $\sim p(X)$.
$p(a)$.
$p(X)$ :- $q(X)$.
$q(b)$.
$q(X)$ :- $p(X)$.

Figure 3 shows a search forest and the corresponding dependency graph among subgoals before the success of $\sim p(c)$ is determined. Notice that $m(c)$ depends upon $p(c)$ negatively due to the node labeled with $m(c)$ :- $\sim p(c)$. A negative edge is marked by a slash in the middle.

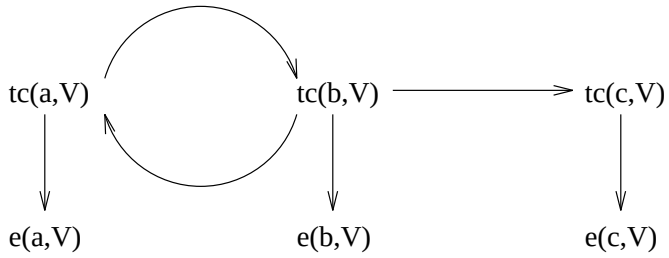


**Figure 2.** Dependency graph for $tc(a, V)$

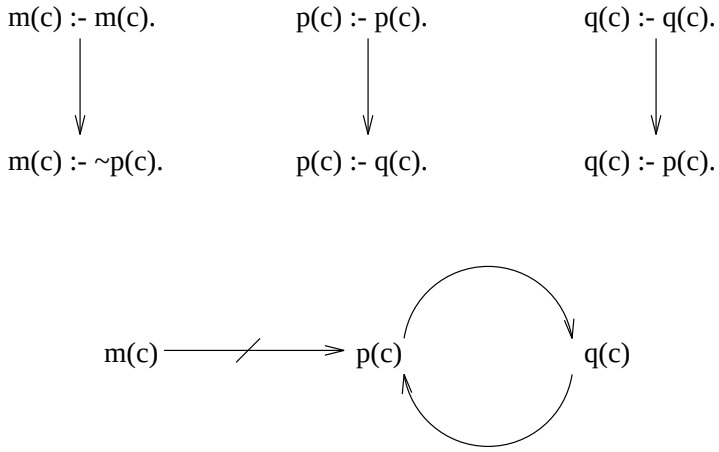


**Figure 3.** Search forest and dependency graph for $m(c)$

As in Prolog, our approach performs a depth-first search and maintains a stack of subgoals. The initial subgoal $m(c)$ is pushed onto the stack first. Traversing the tree for $m(c)$ leads to a new subgoal $p(c)$, which is pushed onto the stack. Traversing the tree for $p(c)$ leads to another subgoal $q(c)$, which is also pushed onto the stack of subgoals. The node $p(c)$ :- $q(c)$ in the tree for $p(c)$ is suspended, waiting for an answer from $q(c)$. Traversing the tree for $q(c)$ leads to a node $q(c)$ :- $p(c)$. Since $p(c)$ has been encountered before and is on the stack, the node $q(c)$ :- $p(c)$ is suspended, waiting for an answer from $p(c)$. The current stack of subgoals is shown in Figure 4.

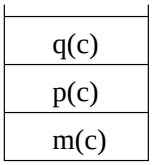


**Figure 4.** Stack of subgoals in a depth-first search for $m(c)$

At this point, there are no new nodes in the tree for $q(c)$ that have not been explored. However, we cannot determine that $q(c)$ is completely evaluated since it

depends upon a previous subgoal, namely $p(c)$, deeper in the stack. Computation returns to subgoal $p(c)$. Similarly, there are no new nodes in the tree for $p(c)$ that have not been explored. But $p(c)$ does not depend upon any previous subgoal deeper in the stack. Furthermore, there are no negative edges among the set $\{p(c), q(c)\}$ of subgoals. Therefore both $p(c)$ and $q(c)$ are completely evaluated, so they are popped off the stack and their suspended clauses are disposed. When $p(c)$ and $q(c)$ are marked as completed, the node waiting on $\sim p(c)$ in the tree for $m(c)$ is processed, and the answer $m(c)$ is derived.

To detect subgoals that are completely evaluated, we maintain, for each subgoal $A$, the deepest subgoal $B$ in the stack which $A$ or any subgoal on top of $A$ may depend upon. When there are no new nodes that have not been explored in the trees for $A$ and subgoals on top of $A$, we check the subgoal associated with $A$. If the subgoal is deeper in the stack than $A$, $A$ may depend upon subgoals below $A$ and therefore cannot be completed. Otherwise, $A$ and all subgoals on top of $A$ are completely evaluated provided that there are no negative edges among these subgoals.

*2.2.3. Negative Loops and Delaying* According to the definition of subgoals that are completely evaluated, every selected ground negative literal from any node in the trees of these subgoals must have been resolved. This may be impossible for programs that are not stratified.

*Example 2.3.* Consider the subgoal $w(a)$ with respect to the following program:

$w(X)$ :- $m(X,Y), \sim w(Y), p(Y)$.
$m(a,b)$. $\quad m(b,c)$. $\quad m(c,b)$.
$p(b)$.

Figure 5 shows the search forest and the dependency graph when a negative loop is encountered. Our implementation follows the depth-first and tuple-at-a-time computation in Prolog, and maintains a stack of subgoals as in Prolog, the current state of which is shown in Figure 6. (A determinacy analysis or indexing scheme may detect that subgoals such as $m(a,Y)$, $m(b,Y)$, and $m(c,Y)$ can be executed without being pushed onto the stack.)

Consider the most recent subgoal $m(c,Y)$. It is called in the node labeled by:

$$w(c) \text{ :- } m(c,Y), \sim w(Y), p(Y)$$

in the tree for $w(c)$ during the evaluation of $w(c)$. Following the tuple-at-a-time strategy, the answer $m(c,b)$ for $m(c,Y)$ is returned immediately to the node waiting on it, which leads to a new node in the tree for $w(c)$:

$$w(c) \text{ :- } \sim w(b), p(b)$$

Since $w(b)$ is on the stack and is not completely evaluated, the new node is suspended. At this point, there are no new nodes in the tree for $m(c,Y)$ that can be explored, nor are there any new nodes created by the answer of $m(c,Y)$ that have not been explored. We check to see if $m(c,Y)$ is completely evaluated. In this example, $m(c,Y)$ does not depend upon any other subgoal. Thus it is completely evaluated and so is popped off the stack and marked as completed. The edge directed towards

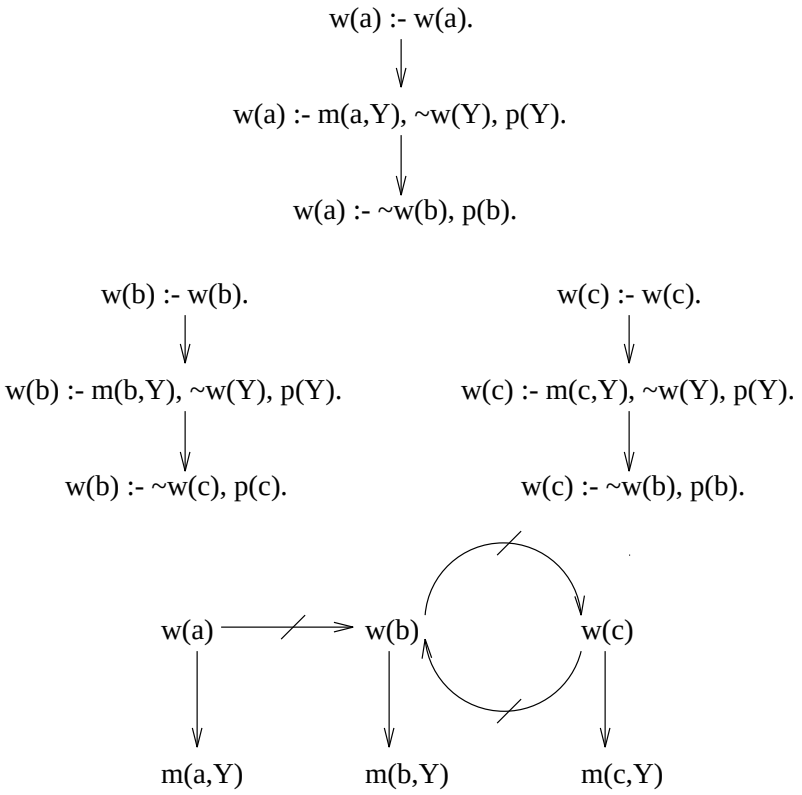


**Figure 5.** Search forest and dependency graph for $w(a)$

$m(c,Y)$ in the dependency graph, namely $w(c)$ :- $m(c,Y), \sim w(Y), p(Y)$, is deleted since all answers of $m(c,Y)$ have been propagated.

Similarly we check the next subgoal on the stack, namely $w(c)$. It cannot be completely evaluated since it depends upon a subgoal, $w(b)$, deeper in the stack. Therefore $w(c)$ remains on the stack.

Computation returns to the next subgoal $m(b,Y)$. The subgoal $m(b,Y)$ does not depend upon other subgoals and is in fact completely evaluated. However without a possibly costly re-organization of the stack, $m(b,Y)$ cannot be popped off due to the fact that $w(c)$ on top of it depends upon a subgoal deeper in the stack than $m(b,Y)$.

Computation then returns to subgoal $w(b)$. No subgoal from the top of the stack up to and including $w(b)$ depends upon any subgoal deeper than $w(b)$. However, neither $w(b)$ nor $w(c)$ can be completed since each depends upon the other through negation.

Our approach in SLG resolution is to delay negative literals in case of possible negative loops so that computation of queries can proceed. It may be the case that another subgoal in the body of the clause may fail, thus in effect eliminating the negative loop. In Figure 5, we delay $\sim w(b)$ in the node:

$$w(c) \text{ :- } \sim w(b), p(b)$$

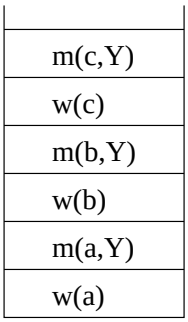


**Figure 6.** Stack of subgoals in a depth-first search for $w(a)$

and the node has a single child labeled by:

$$w(c) \text{ :- } \sim w(b) \mid p(b)$$

Similarly we delay $\sim w(c)$ in the node:

$$w(b) \text{ :- } \sim w(c), p(c)$$

and the node has a single child labeled by:

$$w(b) \text{ :- } \sim w(c) \mid p(c)$$

We use $\mid$ to separate delayed literals (on the left of $\mid$) from the other body literals (on the right of $\mid$) that are yet to be solved. As far as dependencies among subgoals are concerned, delaying eliminates the previous negative edge and possibly introduces a new edge. For instance, delaying $\sim w(b)$ in $w(c)$ :- $\sim w(b), p(b)$ eliminates the corresponding negative edge from $w(c)$ to $w(b)$, and introduces a new edge from $w(c)$ to $p(b)$ by creating a new node $w(c)$ :- $\sim w(b) \mid p(b)$. Figure 7 shows the search forest and the subgoal dependency graph after delaying $\sim w(b)$ and $\sim w(c)$.

Delayed literals are not included in the consideration of subgoal dependencies as far as completely evaluated subgoals are concerned. The intuition is that we are now trying to prove contingent answers, i.e., answers that are implications. So in some sense, the dependency has been moved from the proof into the answer. The delayed literals will, however, have to be simplified if and when their truth or falsity becomes known.

The nodes newly created by delaying are then processed, leading to new subgoals $p(b)$ and $p(c)$ on the stack. Subgoal $p(b)$ succeeds, leading to an answer node:

$$w(c) \text{ :- } \sim w(b) \mid$$

for $w(c)$. Subgoal $p(c)$ fails. Both $p(b)$ and $p(c)$ are completely evaluated and are popped off the stack. Since $w(c)$ does not depend upon any subgoal that is not completed evaluated, $w(c)$ is completely evaluated and popped off the stack, so are $m(b, Y)$ and $w(b)$. However, $w(b)$ is completed without any answers. The failure of $w(b)$ is propagated to the delayed literal $\sim w(b)$ in the answer $w(c)$ :- $\sim w(b) \mid$ ,

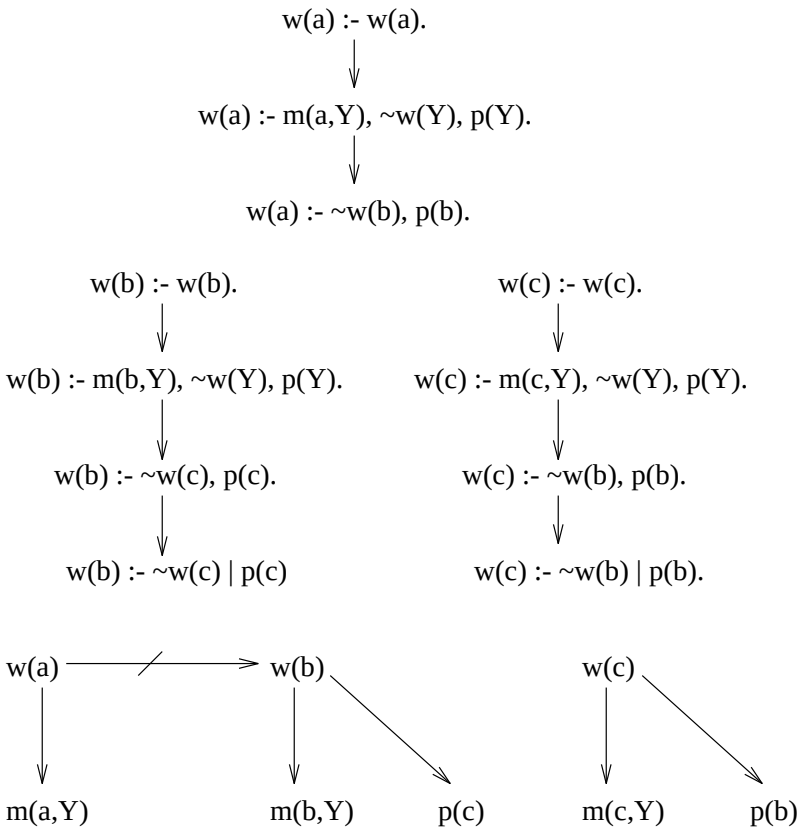


**Figure 7.** Search forest and dependency graph for $w(a)$ after delaying

leading to a definite answer $w(c)$. The failure of $w(b)$ is also propagated to $\sim w(b)$ in the node:

$$w(a) \text{ :- } \sim w(b), p(b)$$

which, after resolving away $p(b)$, leads to an answer $w(a)$.

In general, the well-founded model is three valued. Answers for a subgoal may contain delayed literals that cannot be simplified away, and these answers are neither true nor false in the well-founded model.

## *2.3. Transformations in SLG Resolution*

This subsection reviews the basic definitions and transformations in SLG resolution that are essentially operations over search forests of a query. The correctness theorem of SLG resolution [6, 7] is described and explained, which will be used to establish the correctness of our implementation of SLG resolution.

*Definition 2.3.* An *X-clause* $G$ is a clause of the form:

$$A \text{ :- } D \mid B$$

where $A$ is an atom, $D$ is a sequence of (delayed) ground negative literals and (possibly nonground) atoms, and $B$ is a sequence of literals. Literals in $D$ are called *delayed* literals. If $B$ is empty, an X-clause is called an *answer* clause.

A clause in a program is viewed as an X-clause in which $D$ is empty. We usually omit $\mid$ when $D$ is empty. As far as the declarative semantics is concerned, each X-clause is viewed as an ordinary clause whose body is the conjunction of all literals in $D$ and $B$. That is, the $\mid$ is purely a control annotation.

Given an X-clause $A$ :- $D \mid B$ where $B$ is non-empty, a *computation rule* $R$ selects from $B$ exactly one literal, called the *selected* literal.

*Definition 2.4.* [SLG Resolution] Let $G$ be an X-clause $A$ :- $D \mid L_1, ..., L_n$, where $n > 0$ and $L_i$ be the selected atom. Let $C$ be an X-clause with no delayed literals, and $C'$, of the form $A'$ :- $L'_1, ..., L'_m$, be a variant of $C$ with variables renamed so that $G$ and $C'$ have no variables in common. $G$ is *SLG resolvable* with $C$ if $L_i$ and $A'$ are unifiable. The clause

$$(A \text{ :- } D \mid L_1, ..., L_{i-1}, L'_1, ..., L'_m, L_{i+1}, ..., L_n)\theta$$

is the *SLG resolvent* of $G$ with $C$, where $\theta$ is a most general unifier of $L_i$ and $A'$.

SLG resolution is used for resolution with a clause in a program or with an answer clause that has an empty sequence of delayed literals (on the left of $\mid$ ).

For an answer clause that has a non-empty sequence of delayed literals, relevant variable bindings in the head of the answer clause are propagated by *SLG factoring*, but the sequence of delayed literals in the body is not propagated.

*Definition 2.5.* [SLG Factoring] Let $G$ be an X-clause $A$ :- $D \mid L_1, ..., L_n$, where $n > 0$ and $L_i$ be the selected atom. Let $C$ be an answer clause, and $C'$, of the form $A'$ :- $D' \mid$ , be a variant of $C$ with variables renamed so that $G$ and $C'$ have no variables in common. If $D'$ is not empty and $L_i$ and $A'$ are unifiable with a most general unifier $\theta$, then the *SLG factor* of $G$ with $C$ is

$$(A \text{ :- } D, L_i \mid L_1, ..., L_{i-1}, L_{i+1}, ..., L_n)\theta$$

The motivation of not propagating delayed literals in an answer clause is to guarantee the polynomial complexity for computation of queries on function-free programs[6]. If there are multiple answer clauses with the same atom (up to variable renaming) in the head, only one of them will be propagated by using either SLG resolution or SLG factoring. As far as answer propagation is concerned, two answer clauses are considered distinct if the head atoms are not renaming variants of each other.

We associate with each non-root node in a tree a *status* value, which can be either *new*, *answer*, *active*, *floundered*, or *disposed*. The initial status of each newly created node is *new*. The processing of a *new* node may change the status to:

- *answer* if the clause labeling the node is an answer clause;
- *floundered* if the selected literal is a non-ground negative literal;
- *active* if the selected literal is not floundered and is not completely evaluated; and

- *disposed* if all possible child nodes of the node have been created (and so the node is no longer useful).

Initially, if a query is an atom $A$, the search forest starts with a single tree for $A$, whose root node is labeled $A$ :- $A$ and has a child node for each SLG resolvent of $A$ :- $A$ with program clauses.

Each transformation is an operation that changes the search forest. Transformations (i-iii) process the X-clause of a new node, whose status is changed (mutually exclusively) to *answer, floundered*, or *active*. Transformation (iii) also starts a new subgoal when it is first encountered.

Let $G$ be the X-clause of a new (non-root) node $v$.

(i) NEW ANSWER. If $G$ is an answer clause, then the status of $v$ is changed to *answer*;

If $G$ is not an answer clause, let $L$ be the selected literal of $G$.

(ii) FLOUNDERING. If $L$ is a non-ground negative literal, the status of $v$ is changed to *floundered*;

(iii) NEW ACTIVE. If $L$ is an atom $B$ or a ground negative literal $\sim B$, the status of $v$ is changed to *active* and its associated set of atoms is empty. Furthermore if there is no tree for $B$ in the current search forest, it is created whose root node is labeled with $B$ :- $B$ and has a child node for each SLG resolvent of $B$ :- $B$ with program clauses;

The set of atoms associated with an active node indicates what answers have been returned to the active node.

Let $G$ be the clause of an active node $v$ and $L$ be the selected literal of $G$.

(iv) ANSWER RETURN. If $L$ is an atom $B$ and for some *answer* clause $C$ in the tree for $B$, of the form $H \leftarrow D \mid$ , $H$ is not in the associated set of atoms of $v$, then $H$ is added to the associated set of atoms of $v$, and $v$ has a new child node labeled by the SLG resolvent of $G$ with $C$ on $L$ if $D$ is empty or by the SLG factor of $G$ with $C$ on $L$ if $D$ is not empty;

Transformations (v) and (vi) solve a ground negative subgoal by negation-as-failure if the corresponding positive subgoal is either successful or failed. Otherwise, transformation (vii) delays the selected ground negative subgoal.

If the selected literal $L$ of the X-clause $G$ of an active node $v$ is a ground negative literal $\sim B$, there are three cases:

(v) NEGATION FAILURE-R. If $B$ has an answer with no delayed literals, the status of $v$ is changed to *disposed*;

(vi) NEGATION SUCCESS-R. If $B$ is completely evaluated without any answers, then $v$ has a new child node labeled by $G$ with $L$ deleted, and the status of $v$ is changed to *disposed*;

(vii) DELAYING. Otherwise, $v$ has a new child node labeled by a clause obtained from $G$ by moving $L$ into the sequence of delayed literals, and the status of $v$ is changed to *disposed*.

Subgoals that are completely evaluated can be determined by inspecting their trees in the current search forest according to the following definition.

*Definition 2.6.* Let $P$ be a program and $Q$ be a query atom. Given the search forest at any point of the computation of $Q$ with respect to $P$, and a set $\mathcal{A}$ of subgoals, $\mathcal{A}$ is *completely evaluated* in the search forest if for every subgoal $A \in \mathcal{A}$, the search forest contains the tree for $A$, whose root node is labeled by $A$ :- $A$ and which satisfies the following conditions:

- For each SLG resolvent $G$ of $A$ :- $A$ with a program clause on the $A$ in the body, the root node has a child node labeled by $G$;
- For each non-root node $v$ labeled by a clause $G$ with a selected atom $B$, either $B$ is already marked as completed or $B \in \mathcal{A}$, and for every distinct atom $B'$ that occurs in the head of some answer clause of $B$, $v$ has a child node labeled by the SLG resolvent or SLG factor of $G$ with $C$ on $B$, where $C$ is an answer clause with $B'$ in the head;
- For each non-root node $v$ labeled by a clause $G$ with a selected negative literal $\sim B$, $B$ is ground. Furthermore, either $B$ has an answer $B$ and $v$ is a failed leaf node; or $B$ is already marked as completed and has no answers, in which case $v$ has a single child labeled by $G$ with $\sim B$ deleted; or $\sim B$ is delayed and $v$ has a single child node labeled by $G'$ obtained from $G$ by delaying $\sim B$ (i.e., moving $\sim B$ from the right to the left of the $|$ ).

The completion transformation is as follows:

(viii) COMPLETION. Let $\mathcal{A}$ be a non-empty subset of subgoals that is completely evaluated. Then for each $A \in \mathcal{A}$, every *active* node in the tree for $A$ is *disposed* and $A$ is marked as *completed.*

Given an arbitrary but fixed computation rule, there are programs in which ground negative literals must be delayed before their truth or falsity is known. Additional transformations are needed for simplifying delayed literals when their truth value is determined, the details of which are omitted. These transformations have no effect on the correctness of SLG resolution, but are necessary to derive the most simplified answer clauses.

We also use the term *SLG resolution* to refer to the process of applying transformations starting with the initial forest of a query atom with respect to a program. Since the Herbrand universe is countable, there is a stage, which may be larger than $\omega$, when no transformation can be applied to the search forest of a query. It was shown [6] that when no transformation can be applied to a search forest, either some node in the forest is floundered or every subgoal in the forest is marked as completed. In the latter case, the only nodes that are not disposed in the tree of each subgoal are the root node and the answer nodes. If $A$ is the initial query atom, let $P_A$ denote the set of all answer clauses in the search forest at the end.

The well-founded partial model of a logic program coincides with the smallest three valued stable model [17]. The correctness of SLG resolution is proved in [6] using three valued stable models.

*Theorem 2.1. [6] Let $P$ be a program, $R$ be an arbitrary but fixed computation rule, $A$ be a query atom, and $P_A$ be the set of all answer clauses in the final search forest derived from $A$ that has no floundered nodes. Let $HB$ be the set of all ground instances of all atoms in $P_A$. Then*

*for every three valued stable model $M$ of $P$, the restriction of $M$ to $HB$, denoted by $M|_{HB}$, is a three valued stable model of $P_A$; and*

- *for every three valued stable model $M_A$ of $P_A$, which is an interpretation over $HB$, there exists a three valued stable model $M$ of $P$ such that $M|_{HB} = M_A$.*

*In particular, $WF(P)|_{HB} = WF(P_A)$.*

A key step in the proof of the theorem is to show that each transformation preserves all three valued stable models. Let $P$ be a program. Given any search forest that has been constructed for a query atom $A$ with respect to $P$, the clauses of all non-root nodes that are not disposed in the forest represent a partially evaluated program $P_A$ for all the subgoals in the search forest. The literals on the right of | in each X-clause remain to be evaluated with respect to $P$, while delayed literals on the left of | are partially evaluated. To relate partially evaluated subgoals to the original program, we replace each predicate $p$ in $P_A$ that occurs in the head of an X-clause or in a delayed literal with a new distinct primed predicate $p'$ (of the same arity). Let the resulting program be denoted by $P'_A$. The invariant of the proof is that in every three valued stable model of $P \cup P'_A$, the meaning of each primed atom coincides with that of the corresponding unprimed atom. This invariant holds for the initial forest, and is preserved by each transformation. When every subgoal in a search forest is completely evaluated, $P_A$ contains only answer clauses, and the program $P'_A$ becomes independent of predicates in $P$, which leads to the theorem above. Readers are referred to [6] for further details of the proofs.

## 3. DATA REPRESENTATION AND DEPENDENCY MAINTENANCE

There are two major issues in an efficient implementation of SLG resolution, namely completion and delaying. Completion, if implemented directly according to the definition, requires inspection of the trees of a set of subgoals in order to check whether they are completely evaluated. The cost of checking for completion can become a bottleneck. Delaying basically skips a negative literal so that the rest of the body of an X-clause can be solved. Delaying is needed to handle negative loops, but should be avoided as much as possible in order to reduce computation of subgoals that are irrelevant to a query. This section describes the data representation for a search forest and an incremental scheme for dependency maintenance. The latter is used for efficient completion and negative loop checking.

### *3.1. Table Entries*

The search forest is represented by a global table $\mathcal{T}$ of subgoals. Each table entry is identified by a subgoal, and is of the form $(A, Anss, Poss, Negs, Comp)$, where

- $A$ is a subgoal;
- $Anss$ is the set of answers in the current tree for $A$;
- $Poss$ is a sequence of pairs $(B, G)$, where $B$ is a subgoal and $G$ is an X-clause labeling an active node in the tree for $B$ with the selected atom $A$;
- $Negs$ is a sequence of pairs $(B, G)$, where $B$ is a subgoal and $G$ is an X-clause labeling an active node in the tree for $B$ with the selected ground negative literal $\sim A$;
- $Comp$ is a boolean variable indicating whether $A$ is completely evaluated.

In a pair $(B, G)$, $B$ is the subgoal that is waiting on $A$ through an edge represented by the clause $G$. Whenever an answer for $A$ is found, it is returned to every pair $(B, G)$ that is waiting in $Poss$ or $Negs$. Thus there is no need to have an explicit representation of the set of all answers that have been returned to a waiting node. We use $Anss(A)$, $Poss(A)$, $Negs(A)$, and $Comp(A)$ to denote the corresponding fields of $A$ in table $\mathcal{T}$.

In our implementation, each new node is processed immediately so that its status is changed to either *answer*, *active*, *disposed*, or *floundered*. Upon floundering, the computation halts with an error message. Therefore only clauses of answer nodes and active nodes have to be represented in a table.

## *3.2. Dependency Maintenance: A Simple Scheme*

A stack of subgoals is used to maintain dependencies. They are updated incrementally whenever an edge from one subgoal to another is processed, and are checked at certain points for completion and delaying.

*3.2.1. Stack Entries* For smooth integration with Prolog, the search forest of a query is traversed in a depth-first manner using a left-most computation rule. A stack $\mathcal{S}$ of subgoals is maintained, which is similar to the local stack in Prolog.

New subgoals that are encountered during a depth-first search are pushed onto the stack. Each subgoal has an associated *depth-first number* (DFN) so that the relative position of two subgoals in the stack is determined easily by comparing their DFNs. We say that a subgoal $A$ is on top of another subgoal $B$ (or $B$ is below $A$) if both $A$ and $B$ are on the stack and $A$ is pushed onto the stack after $B$. A global counter (COUNT) is used to compute the next depth-first number. It is initialized to 1.

The stack $\mathcal{S}$ plays an important role in detecting completely evaluated subgoals and potential negative loops. The basic idea is as follows.

When a new subgoal $A$ is encountered, it is pushed onto $\mathcal{S}$. A depth-first traversal of the tree for $A$ is initiated, which may lead to other new subgoals that are pushed onto the stack after $A$.

We associate with each subgoal $A$ two additional numbers, called *PosLink* and *NegLink*, respectively. PosLink is initialized to the depth-first number of $A$, and NegLink is initialized to *maxint* – a value that is larger than all possible depth-first numbers in an implementation. For each subgoal $A$, we denote by PosLink($A$) and NegLink($A$) the corresponding PosLink and NegLink of $A$. The stack entry in $\mathcal{S}$ for subgoal $A$ is of the form ($A$,DFN,PosLink,NegLink).

The PosLink of a subgoal $A$ captures the deepest subgoal on the stack which $A$ may depend upon through positive edges, and the NegLink of $A$ represents the deepest subgoal on the stack which $A$ may depend upon through at least one negative edge. The PosLink and NegLink of $A$ are updated when an edge originating from $A$ is explored.

*3.2.2. Incremental Updates of Dependencies* Suppose that the tree for $A$ has a non-root node $v$ labeled by an X-clause $G$ with a selected atom $L$.

Assume that $L$ is an atom $B$. If $B$ is not a new subgoal and is not completed, $B$ must be on the stack. The PosLink and NegLink of $A$ are updated by the following assignments:

$$\text{PosLink}(A) := \min(\text{PosLink}(A), \text{PosLink}(B))$$
$$\text{NegLink}(A) := \min(\text{NegLink}(A), \text{NegLink}(B))$$

where 'min' is the function that returns the minimum value of all its arguments.

If $B$ is a new subgoal, a depth-first traversal of the tree for $B$ is initiated. When it finishes, if $B$ is completely evaluated, all answers of $B$ must have been returned to the X-clause $G$. This is due to the tuple-at-a-time strategy in which we return each answer immediately to every waiting node. In this case, the PosLink and NegLink of $A$ are *not* updated. If $B$ is not completely evaluated, then $B$ must be on the stack, in which case PosLink and NegLink of $A$ are updated as above.

Another possibility is that the selected literal $L$ is a ground negative literal $\sim B$. Then there is a negative edge from $A$ to $B$.

If $B$ is not a new subgoal and is not completed, $B$ must be on the stack. If $B$ has a definitely true answer, $v$ is marked as a failed leaf node. The PosLink and NegLink of $A$ are not updated in this case as the success of $B$ has been propagated. If $B$ has not succeeded with a definitely true answer, the PosLink of $A$ is left unchanged, but the NegLink of $A$ is updated as follows:

$$\text{NegLink}(A) := \min(\text{NegLink}(A), \text{PosLink}(B), \text{NegLink}(B))$$

If $B$ is a new subgoal, a depth-first traversal of the tree for $B$ is started. When it returns, if $B$ is completely evaluated, it must have been popped off the stack, and our strategy processes every node waiting on $B$ or $\sim B$ when $B$ is marked as completed. The PosLink and NegLink of $A$ are not updated. If $B$ is not completed, $B$ must be still on the stack, and the same update is carried out for the NegLink of $A$.

*3.2.3. Checking for Completion and Delaying* When the depth-first traversal of the tree for $A$ finishes, we check the PosLink and NegLink of $A$.

- If $\text{PosLink}(A) = \text{DFN}(A)$ and $\text{NegLink}(A) = \text{maxint}$, then $A$ and all subgoals on top of $A$ are completely evaluated. They are marked as completed and are popped off the stack. All nodes waiting on any of these subgoals or its negation are processed.
- If $\text{PosLink}(A) = \text{DFN}(A)$ and $\text{DFN}(A) \leq \text{NegLink}(A) < \text{maxint}$, then there may be negative loops among $A$ and subgoals on top of $A$, in which case delaying should be applied.
- Otherwise, $A$ or some subgoal on top of $A$ depends upon some subgoal deeper in the stack than $A$. They remain on the stack and are not marked as completed. Computation returns to the subgoal immediately below $A$ on the stack.

## *3.3. Problems with the Simple Scheme*

The checking for completion in the simple scheme assumes implicitly that every subgoal depends upon all subgoals on top of it. That is, when $\text{PosLink}(A) = \text{DFN}(A)$ and $\text{NegLink}(A) = \text{maxint}$, both $A$ and all subgoals on top of $A$ are considered to be completely evaluated. However, the PosLink and NegLink of each subgoal captures only explicit dependencies from edges between subgoals. As a result, the simple scheme does not work in general. Some subgoal $C$ may be

pushed onto the stack on top of $B$ even though there is no path from $B$ to $C$. Furthermore $C$ may depend upon subgoals below $B$ on the stack. Therefore when $B$ becomes completely evaluated, the subgoal $C$ on top of $B$ is popped off the stack as well, which could be wrong. This can happen when an answer is returned to a node that has a selected atom or when the selected ground negative literal of a node is resolved.

*3.3.1. Answer Return to a Positive Literal* Suppose that there is a non-root node in the tree for $A$ labeled by an X-clause $G$ of the form:

$$A' \text{ :- } D \mid B, C$$

with the selected atom $B$, and $B$ is a new subgoal. According to the tuple-at-a-time strategy, as soon as an answer for $B$ is derived, it is returned to the waiting clause $G$, and the next subgoal $C$, which happens to be a new subgoal too, is processed. Therefore $C$ is on top of $B$ and $B$ is on top of $A$ on the stack, even though there is no dependency between $B$ and $C$ at all. The following example illustrates this situation.

*Example 3.1.* Consider the following program and query $p$:

$p$ :- $q, r$.
$p$.
$q$.
$r$ :- $p$.

The dependency graph for$p$ is depicted in Figure 1 (a). The initial subgoal $p$ is pushed onto the stack, whose entry is $(p, 1, 1, maxint)$. The evaluation of $p$ leads to a new subgoal $q$, whose stack entry is $(q, 2, 2, maxint)$. By the tuple-at-a-time computation, the answer $q$ is returned immediately to the node labeled by:

$$p \text{ :- } q, r$$

A new node:

$$p \text{ :- } r$$

is created and is expanded immediately. The rule matching $r$ generates an edge from $r$ to $p$, which is below $q$ on the stack. Thus the PosLink of $r$ is updated to 1. When there are no new nodes to be explored, computation returns to the most recent subgoal, which is $r$. The current stack of subgoals is shown in Figure 1 (b).

Subgoal $r$ is not completely evaluated since it depends upon $p$ deeper in the stack, and so PosLink($r$) < DFN($r$). When the PosLink and NegLink of $q$ are checked, we have that PosLink($q$) = DFN($q$) and NegLink($q$) = maxint. According to the simple scheme, $q$ and the subgoal on top of it, namely $r$, are completely evaluated. This is clearly wrong since $r$ should have an answer from $p$ when the second clause of $p$ is explored.

Completion is not required for query evaluation with respect to positive programs, but it can help reusing the stack space by popping off subgoals that are completely evaluated. Example 1 shows that the simple scheme does not work for positive programs.

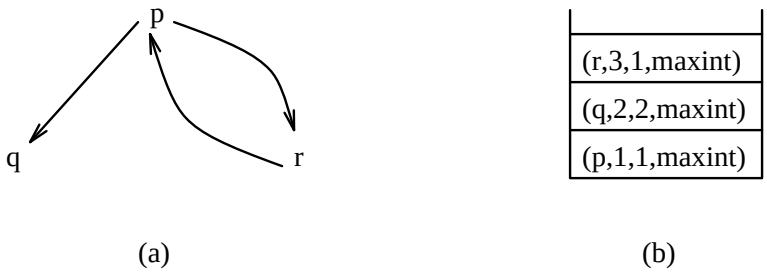


**Figure 1.** Dependency graph and stack of subgoals in a depth-first search for $p$

*3.3.2. Success of a Negative Literal* The success of a ground negative literal can also lead to subgoals on top of a subgoal $A$, even though there may be no dependencies between them.

*Example 3.2.* Consider query $p$ with respect to the following program:

$p$ :- $q, \sim c, r$.
$p$.
$q$.
$r$ :- $p$.

The program is a slight variant of that in Example 1. The dependency graph of subgoals is shown in Figure 2(i).

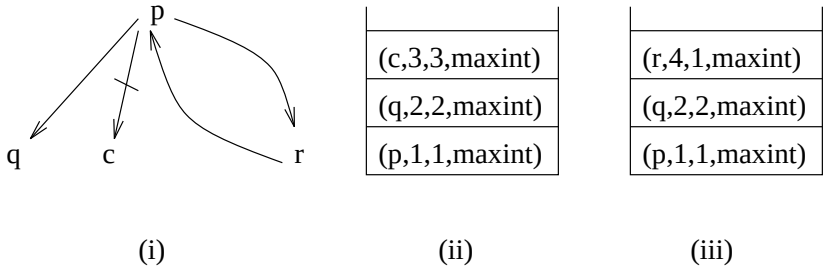


**Figure 2.** Dependency graph and stacks for $p$

Figure 2(ii) shows the stack of subgoals when $c$ is being processed. Since there is no clause for $c$, $c$ is completely evaluated without any answers and is popped off the stack. Therefore the negative literal $\sim c$ succeeds, which leads to a new subgoal $r$. Figure 2(iii) shows a situation similar to that in Example 1.

## *3.4. Dependency Maintenance: A Correct Scheme*

The simple scheme assumes implicit dependencies of a subgoal upon all subgoals on top it on the stack when it checks for completion. We modify the simple scheme to capture the implicit dependencies and describe how dependencies are updated when negative loops are handled.

*3.4.1. Capturing Implicit Dependencies* We modify the procedure for the depth-first computation of a subgoal. The depth-first computation for $A$ returns two numbers, called *PosMin* and *NegMin*, respectively. While the PosLink and NegLink of $A$ capture the direct dependencies through edges coming out of $A$ in the dependency graph, the PosMin and NegMin returned from the evaluation of $A$ also model the implicit dependencies by the linear nature of the stack of subgoals as illustrated in Example 1 and Example 2. In other words, the PosMin of $A$ is the minimum depth-first number of all subgoals which $A$ and subgoals on top of $A$ on the stack may depend upon through positive edges, and the NegMin of $A$ is the minimum depth-first number of all subgoals which $A$ and subgoals on top of $A$ may depend upon through some negative edges.

When the depth-first computation of $A$ finishes, PosMin and NegMin are first merged with PosLink and NegLink of $A$, i.e.,

$$\text{PosLink}(A) := \min(\text{PosLink}(A), \text{PosMin})$$
$$\text{NegLink}(A) := \min(\text{NegLink}(A), \text{NegMin})$$

The same method is then used to determine if $A$ and subgoals on top of it are completely evaluated or may be involved in negative loops. The effect is that the completion of $A$ is postponed until all subgoals on top of $A$ are also completely evaluated.

*3.4.2. Dependency Update after Delaying* Let $S_A$ be the set of subgoals from the top of the stack $\mathcal{S}$ down to and including $A$. Suppose that $\text{PosLink}(A) = \text{DFN}(A)$ and $\text{DFN}(A) \leq \text{NegLink}(A) < \text{maxint}$, which indicates that there may be negative loops among subgoals in $S_A$. The DELAYING transformation is applied to every node $v$ in the current search forest such that $v$ is labeled by an X-clause with a selected ground negative literal $\sim B$, where $B$ is in $S_A$. As far as the dependency graph is concerned, all negative edges to subgoals in $S_A$ are eliminated. This is reflected by resetting NegLink of every subgoal in $S_A$ to maxint. Subgoals in $S_A$ remain on the stack and will be re-checked again after all the new nodes created by DELAYING are processed.

*Example 3.3.* Consider the following program and a subgoal $s$.

$$s \text{ :- } \sim p, \sim q.$$
$$p \text{ :- } \sim s, q.$$
$$q \text{ :- } \sim s, p.$$

Initially, the subgoal is $s$, and $(s, 1, 1, maxint)$ is pushed onto the stack of subgoals.

Traversing the tree for $s$ leads to a new subgoal $p$, and so $(p, 2, 2, maxint)$ is pushed onto the stack. The node, $p$ :- $\sim s, q$, represents a negative edge from $p$ to $s$. Therefore the NegLink of $p$ is updated to 1. The node, $p$ :- $\sim s, q$, is suspended, and computation returns to $s$. The NegLink of $s$ is updated to the minimum of PosLink and NegLink of $p$, which is 1. Figure 3 shows the search forest, the dependency graph, and the stack of subgoals at this point.

Since $\text{PosLink}(s) = \text{DFN}(s)$ and $\text{DFN}(s) \leq \text{NegLink}(s) < \text{maxint}$, $\{p, s\}$ may be (and, in this case, are) involved in negative loops. We apply the DELAYING transformation to all the negative edges with a selected ground negative literal $\sim p$ or $\sim s$. This creates two new nodes, namely $s$ :- $\sim p \mid \sim q$ and $p$ :- $\sim s \mid q$. In

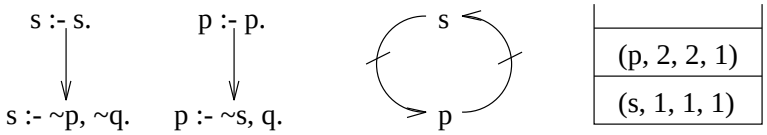


**Figure 3.** The first negative loop for $s$

effect, the two negative edges in Figure 3 are eliminated. The NegLink of $s$ and the NegLink of $p$ are both reset to $maxint$. Computation continues by exploring the newly created nodes and then $p$ and $s$ will be re-checked for completion.

Exploring the node, $s$ :- $\sim p \mid \sim q$, leads to a new subgoal $q$, and so $(q, 3, 3, maxint)$ is pushed onto the stack. Traversing the tree for $q$ leads to a node $q$ :- $\sim s, p$. The NegLink of $q$ is updated to 1. Since NegLink($q$) < DFN($q$), $q$ is not completely evaluated and remains on the stack. After the traversal of the tree for $q$ finishes, the NegLink of $s$ is updated to 1 since there is a negative edge from $s$ to $q$ and the minimum of the PosLink and NegLink of $q$ is 1.

Computation continues to explore the node, $p$ :- $\sim s \mid q$. The NegLink of $p$ is updated to 1 since NegLink of $q$ is currently 1. As NegLink($p$) < DFN($p$), $p$ remains on the stack. Figure 4 shows the search forest, the dependency graph, and the stack of subgoals at this point.

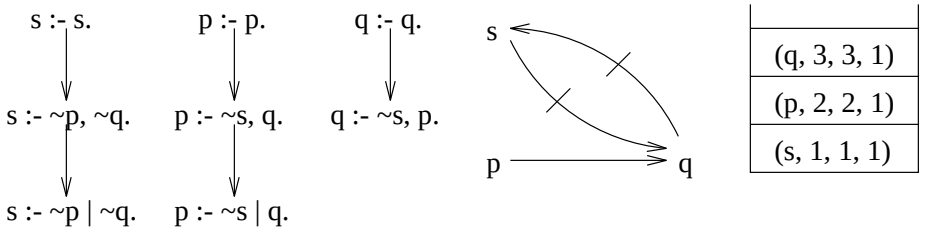


**Figure 4.** The second negative loop for $s$

We check the dependencies of $s$. It holds that PosLink($s$) = DFN($s$) and DFN($s$) $\leq$ NegLink($s$) < maxint. The dependency graph reveals that there are negative loops among $\{q, p, s\}$. DELAYING transformation is applied and the NegLinks of $q$, $p$ and $s$ are reset to $maxint$.

The new node, $s$ :- $\sim p, \sim q \mid$ , is an answer node since there are no literals on the right of $\mid$ . Subgoal $s$ no longer depends upon any other subgoal in the dependency graph, although delayed literals will have to be simplified if and when their truth or falsity becomes known. The new node, $q$ :- $\sim s \mid p$, is explored, and the PosLink of $q$ is updated to the PosLink of $p$, which is 2. Figure 5 shows the search forest, the dependency graph, and the stack of subgoals at this point, where the isolated node $s$ is not displayed in the dependency subgoal.

Notice that PosLink($p$) = DFN($p$) and NegLink($p$) = maxint. Both $q$ and $p$ are completely evaluated and are popped off the stack. The failure of $q$ and $p$ is used to simplify $s$ :- $\sim p, \sim q \mid$ , deriving a definitely true answer for $s$. Similarly $s$ is completely evaluated and popped off the stack, and thus computation of the initial subgoal $s$ terminates.

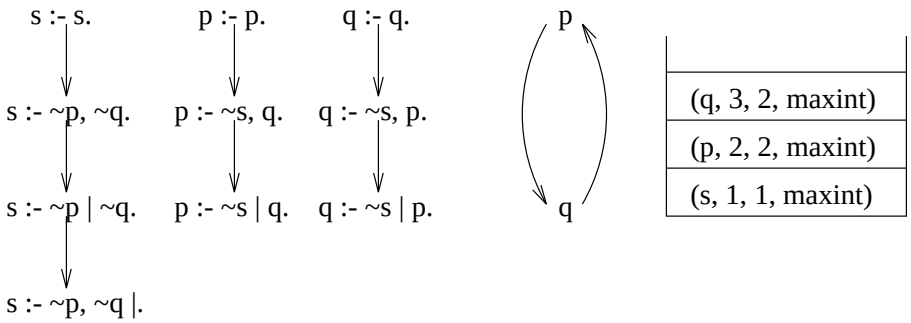


**Figure 5.** After elimination of the second negative loop

## 4. ALGORITHM

This section describes in detail the mutually recursive procedures for an implementation of SLG resolution. We separate them into two groups, one for basic transformations and the other for COMPLETION transformation. We establish the correctness of the implementation by relating it to the correctness of SLG resolution.

### *4.1. Basic Transformations*

Let $P$ be an arbitrary logic program, and $R$ be an arbitrary but fixed computation rule. Without loss of generality, we assume that the initial query consists of only one atom. For each subgoal $A$, $K_A$ denotes the set of clauses in $P$ with which $A$ :- $A$ is SLG resolvable.

Three global variables are used, namely the table $\mathcal{T}$ of subgoals, the stack $\mathcal{S}$ of subgoals, and a counter (COUNT), which have been described in Section 3. Figure 1 shows the main program. It initializes COUNT to 1, inserts a table entry for the initial subgoal $A$ into the table, and pushes an entry $(A, 1, 1, maxint)$ of the initial subgoal $A$ onto the stack. COUNT is incremented every time a new subgoal is pushed onto the stack. After initialization, the main program calls SLG_SUBGOAL to carry out a depth-first computation of subgoal $A$. Pushing an entry onto the stack and calling SLG_SUBGOAL corresponds to the creation of a tree for a new subgoal in the search forest.

In SLG_SUBGOAL($A$,PosMin,NegMin), $A$ is a new subgoal that has just been inserted into the table $\mathcal{T}$ and pushed onto the stack $\mathcal{S}$. PosMin and NegMin are input/output variables. As discusses in Section 3, PosMin and NegMin return the minimum depth first number of all subgoals which $A$ or subgoals on top of $A$ may depend upon through positive edges only and through at least one negative edge respectively. They are passed through all recursive procedures that will be called during the execution of SLG_SUBGOAL($A$,PosMin,NegMin).

Procedure SLG_SUBGOAL creates a new node for each child of the root node of a subgoal. For each newly created non-root node, SLG_NEWCLAUSE is called to process the new node, or more precisely the X-clause labeling the new node. The processing may lead to other new nodes or even new subgoals, which are handled recursively by calling other procedures. Therefore each procedure implements not just one transformation, but a sequence of transformations. When SLG_NEWCLAUSE

```
Input:      a program P and a query atom A.
Output:     a set of answer clauses.
Algorithm:
    begin
        Initialize Count to be 1;
        Initialize T to be the table with one entry, (A,{},[],[],false);
        Initialize S to be the empty stack of subgoals;
        DFN := Count; PosLink := DFN; NegLink := maxint;
        push (A,DFN,PosLink,NegLink) onto stack S;
        Count := Count+1;
        PosMin := DFN; NegMin := maxint;
        SLG_SUBGOAL(A,PosMin,NegMin);
        output all answer clauses in T;
    end
```

**Figure 1.** Algorithm for SLG resolution

returns for all the child nodes of the root node of a subgoal, SLG_COMPLETE is called to determine if $A$ and its relevant subgoals are completely evaluated. Figure 2 shows the details of the procedure SLG_SUBGOAL and the procedure SLG_NEWCLAUSE.

In procedure SLG_NEWCLAUSE($A$,$G$,PosMin,NegMin), $G$ is an X-clause labeling a new non-root node $v$ in the tree for subgoal $A$. SLG_NEWCLAUSE calls procedures SLG_ANSWER, SLG_POSITIVE or SLG_NEGATIVE, depending upon whether the newly created X-clause $G$ has no selected literal, a positive selected literal, or a ground negative selected literal. The branching corresponds to transformations NEW ANSWER, NEW ACTIVE, and FLOUNDERING, which changes the status of the new node $v$.

Procedure SLG_ANSWER (see Figure 3) checks to see if an answer for $A$ is new. If the answer is not subsumed by any existing answer for $A$, SLG_ANSWER proceeds to apply all transformations ANSWER RETURN and NEGATION FAILURE-R that are made possible by the new answer. In particular, if the answer has an empty body, all (active) nodes that are waiting on $\sim A$ are failed and disposed. The answer is returned to *all* nodes waiting on $A$ by either SLG resolution or SLG factoring. All new nodes created by these transformations are handled by calling SLG_NEWCLAUSE recursively.

In procedure SLG_POSITIVE($A$, $G$, $B$, PosMin, NegMin) shown in Figure 4, $G$ is an X-clause labeling an active non-root node in the tree for $A$ and has a selected atom $B$. Therefore $G$ is a positive edge from $A$ to $B$. If $B$ is not in the table $\mathcal{T}$, then $B$ must be a new subgoal and so $G$ is a *solution* edge from $A$ to $B$ (as $G$ leads to the creation of a new subgoal $B$). The new subgoal $B$ is inserted into the table and pushed onto the stack. Notice that the pair $(A, G)$ is inserted into the positive waiting list for answers of $B$. A depth-first computation is initiated for $B$ by calling SLG_SUBGOAL, which returns BPosMin and BNegMin. BPosMin and BNegMin represent the minimum depth-first number of all subgoals that $B$ and subgoals on top of $B$ may depend upon through positive edges only and through at least some

```
procedure SLG_SUBGOAL(A,PosMin,NegMin):
begin
    for each SLG resolvent G of A :- A with some clause C ∈ K_A do begin
        SLG_NEWCLAUSE(A,G,PosMin,NegMin);
    end;
    SLG_COMPLETE(A,PosMin,NegMin);
end

procedure SLG_NEWCLAUSE(A,G,PosMin,NegMin);
begin
    if G has no body literal on the right of | then
        SLG_ANSWER(A,G,PosMin,NegMin)
    else if G has a selected atom B then
        SLG_POSITIVE(A,G,B,PosMin,NegMin)
    else if G has a selected ground negative literal ~B then
        SLG_NEGATIVE(A,G,B,PosMin,NegMin)
    else begin /* G has a selected non-ground negative literal */
        halt with an error message
    end
end
```

**Figure 2.** Procedures to evaluate a subgoal

negative edges respectively. The PosLink and NegLink of $A$ and the corresponding PosMin and NegMin are then updated by calling procedure UPDATE_SOLUTION.

If $B$ is in the current table $\mathcal{T}$, then $B$ is not a new subgoal and so $G$ is a *lookup* edge from $A$ to $B$. If $B$ is not marked as completed, we insert $(A, G)$ into the positive waiting list for potentially more answers of $B$ so that this node will be notified if more answers are added for $B$. The PosLink and NegLink of $A$ and the corresponding PosMin and NegMin are updated by calling UPDATE_LOOKUP. This procedure effectively applies ANSWER RETURN transformations to return any existing answers of $B$ to the node labeled by $G$ in the tree for subgoal $A$. The resolution of these answers creates new nodes in the tree for subgoal $A$ each of which is processed in its turn by the procedure SLG_NEWCLAUSE.

The procedure SLG_NEGATIVE($A$, $G$, $B$, PosMin, NegMin), shown in Figure 5, handles an active node in the tree for $A$ that is labeled by an X-clause $G$ with a selected ground negative literal $\sim B$. Its structure is similar to that of SLG_POSITIVE.

If $B$ is not in the table $\mathcal{T}$, then $B$ must be a new subgoal and $G$ is a *solution* edge from $A$ to $B$. It is inserted into the table and pushed onto the stack. The pair $(A, G)$ is inserted in the negative waiting list of $B$, waiting for the truth value of $B$ to be determined. A depth-first computation of $B$ is initiated by calling SLG_SUBGOAL. It returns BPosMin and BNegMin that represent the minimum depth-first number of all subgoals that $B$ and subgoals on top of $B$ may depend up through positive edges only and through at least some negative edges respectively. When the depth-first computation of $B$ finishes, the PosLink and NegLink

```
procedure SLG_ANSWER(A,G,PosMin,NegMin):
begin
    if G is not subsumed by any answer in Anss(A) in T then begin
        insert G into Anss(A);
        if G has no delayed literals then begin
            reset Negs(A) to empty;
            let L be the list of all pairs (B, H'), where (B, H) ∈ Poss(A) and
                H' is the SLG resolvent of H with G;
            for each (B, H') in L do begin
                SLG_NEWCLAUSE(B,H',PosMin,NegMin);
            end;
        end else begin /* G has a non-empty delay. */
            if no other answer in Anss(A) has the same head as G does then
            begin
                let L be the list of all pairs (B, H'), where (B, H) ∈ Poss(A)
                    and H' is the SLG factor of H with G;
                for each (B, H') in L do begin
                    SLG_NEWCLAUSE(B,H',PosMin,NegMin);
                end;
            end;
        end;
    end;
end
```

**Figure 3.** Procedure to handle an answer node

of $A$ and the corresponding PosMin and NegMin are updated by calling procedure UPDATE_SOLUTION.

If $B$ is already in the table $\mathcal{T}$, then $B$ is not a new subgoal and $G$ is a *lookup* edge from $A$ to $B$. If $B$ is not yet marked as completed, we check if $B$ has a definitely true answer. If so, the node labeled by $G$ in the tree for subgoal $A$ is a failed leaf node by transformation NEGATION FAILURE-R. Otherwise, $(A, G)$ is inserted into the negative waiting list of $B$. The PosLink and NegLink of $A$ and the corresponding PosMin and NegMin are updated by calling UPDATE_LOOKUP. If $B$ is already marked as completed, either NEGATION SUCCESS-R is applied if $B$ has no answers, or DELAYING is applied if $B$ has only answers with delayed literals. In either case, the new node is then processed by calling SLG_NEWCLAUSE.

It should be mentioned that all basic transformations are applied in an eager manner, which is ensured by the following properties of the implementation.

- For any new subgoal $A$ that is encountered, including the initial one, all the child nodes of the root node of the tree for $A$ are created by SLG resolution with program clauses. Every newly created node is processed immediately by SLG_NEWCLAUSE.
- Each new answer of a subgoal $A$ is returned immediately to every active node with a selected atom $A$ as soon as the answer is found. In addition, if the answer does not have delayed literals, all active nodes waiting on $\sim A$

```
procedure SLG_POSITIVE(A,G,B,PosMin,NegMin):
begin
    if B is not in table 𝒯 then begin
        insert (B, {}, [(A, G)], [], false) into 𝒯;
        DFN := Count; PosLink := Count; NegLink := maxint;
        push (B,DFN,PosLink,NegLink) onto stack 𝒮;
        Count := Count+1;
        BPosMin := DFN; BNegMin := maxint;
        SLG_SUBGOAL(B,BPosMin,BNegMin);
        UPDATE_SOLUTION(A,B,pos,PosMin,NegMin,BPosMin,BNegMin);
    end else begin
        if Comp(B) is not true then begin
            insert (A, G) into Poss(B);
            UPDATE_LOOKUP(A,B,pos,PosMin,NegMin);
        end;
        let L be the empty list;
        for each atom B' in the head of some answer in Anss(B) do begin
            if B' :- | ∈ Anss(B) then begin
                let G' be the SLG resolvent of G with B' :- | ;
                insert (A, G') into L;
            end else begin
                let H ∈ Anss(B) with head atom B';
                let G' be the SLG factor of G with H;
                insert (A, G') into L;
            end;
        end;
        for each (A, G') in L do begin
            SLG_NEWCLAUSE(A,G',PosMin,NegMin);
        end;
    end;
end
```

**Figure 4.** Procedure to handle a node with a selected atom

```
procedure SLG_NEGATIVE(A,G,B,PosMin,NegMin):
begin
    if B is not in table 𝒯 then begin
        insert (B, {}, [], [(A, G)], false) into 𝒯;
        DFN := Count; PosLink := DFN; NegLink := maxint;
        push (B,DFN,PosLink,NegLink) onto stack 𝒮;
        Count := Count+1;
        BPosMin := DFN; BNegMin := maxint;
        SLG_SUBGOAL(B,BPosMin,BNegMin);
        UPDATE_SOLUTION(A,B,neg,PosMin,NegMin,BPosMin,BNegMin);
    end else begin
        if Comp(B) is not true then begin
            if B :- | ∉ Anss(B) then begin
                insert (A, G) into Negs(B);
                UPDATE_LOOKUP(A,B,neg,PosMin,NegMin);
            end;
        end else begin
            if Anss(B) = {} then begin
                let G' be G with ~B deleted;
                SLG_NEWCLAUSE(A,G',PosMin,NegMin);
            end else if B :- | ∉ Anss(B) then begin
                let G' be G with ~B delayed;
                SLG_NEWCLAUSE(A,G',PosMin,NegMin);
            end;
        end;
    end;
end
```

**Figure 5.** Procedure to handle a node with a selected ground negative literal

are marked as failed and disposed.

- When an active node $v$ with a selected atom $A$ is encountered, all existing answers for $A$ are returned to $v$. The node $v$ becomes a waiting node for potentially more answers of $A$ (if $A$ is not yet completely evaluated).
- When an active node $v$ with a selected ground negative literal $\sim B$ is processed, $v$ is failed if $B$ has a definitely true answer. If $B$ is completely evaluated with no answers, NEGATION SUCCESS-R is applied.

The operation of each basic transformation is straightforward and corresponds directly to the definition in Section 2.3.

## *4.2. Completion and Delaying*

The application of COMPLETION and DELAYING is carefully controlled in order to ensure an efficient implementation. The DELAYING of a negative literal $\sim B$ is applied under two situations. One is when $B$ is already completed and has only indefinite answers so that it is neither successful nor failed. The other is when there is a potential negative loop.

All edges between subgoals are processed in either SLG_POSITIVE or SLG_NEGATIVE. Let $G$ be an X-clause labeling a non-root node in the tree for subgoal $A$, representing an edge from $A$ to a subgoal $B$. Let Sign be either *pos*itive or *neg*ative representing the polarity of the edge.

If $B$ is already in the table $\mathcal{T}$ and thus is not a new subgoal, $G$ is a *lookup* edge from $A$ to $B$. The PosLink and NegLink of $A$ on the stack are updated using the PosLink and NegLink of $B$ in UPDATE_LOOKUP($A$,$B$,Sign,PosMin,NegMin), and so are PosMin and NegMin.

If $B$ is not yet in the table and thus is a new subgoal, $G$ is a *solution* edge from $A$ to $B$. In this case, a depth-first computation of $B$ is initiated. When it returns, UPDATE_SOLUTION is called to update the PosLink and NegLink of $A$ and the corresponding PosMin and NegMin. The evaluation of $B$ may have left additional subgoals on the stack, which are on top of $A$. They may depend upon subgoals deeper in the stack, which are captured by BPosMin and BNegMin in

UPDATE_SOLUTION($A$, $B$, Sign, PosMin, NegMin, BPosMin, BNegMin)

If $B$ is completely evaluated, BPosMin and BNegMin are propagated. Otherwise, dependencies are updated as in UPDATE_LOOKUP. In the latter case, BPosMin and BNegMin are merged into the PosLink and NegLink of $B$ before the computation of $B$ returns, and so they are implicitly propagated through the PosLink and NegLink of $B$.

Recall that in SLG_SUBGOAL($A$, PosMin, NegMin), SLG_NEWCLAUSE is called for each SLG resolvent $G$ of $A$ :- $A$ with a program clause. Each call to SLG_NEWCLAUSE processes the node labeled by $G$, as well as all new nodes and all new subgoals that are created from the processing of $G$ by calling itself and other procedures recursively. When this is finished, SLG_COMPLETE($A$,PosMin,NegMin) (shown in Figure 7 is called within SLG_SUBGOAL($A$,PosMin,NegMin).

First, PosMin and NegMin are merged with PosLink and NegLink of subgoal $A$ respectively. This is necessary as shown by Example 1 and Example 2 in Section 3.

If PosLink($A$) = DFN($A$) and NegLink($A$) = maxint, then $A$ and all subgoals on top of $A$ are considered to be completely evaluated and are popped off the stack. All

```
procedure UPDATE_LOOKUP(A,B,Sign,PosMin,NegMin):
begin
    if Sign= pos then begin
        PosLink(A) := min(PosLink(A),PosLink(B));
        NegLink(A) := min(NegLink(A),NegLink(B));
        PosMin := min(PosMin, PosLink(B));
        NegMin := min(NegMin, NegLink(B));
    end else begin /* Sign = neg */
        NegLink(A) := min(NegLink(A),PosLink(B),NegLink(B));
        NegMin := min(NegMin, PosLink(B), NegLink(B));
    end;
end

procedure UPDATE_SOLUTION(A,B,Sign,PosMin,NegMin,BPosMin,BNegMin):
begin
    if Comp(B) ≠ true then begin
        UPDATE_LOOKUP(A,B,Sign,PosMin,NegMin);
    else begin
        PosLink(A) := min(PosLink(A), BPosMin);
        NegLink(A) := min(NegLink(A),BNegMin);
        PosMin := min(PosMin, BPosMin);
        NegMin := min(NegMin, BNegMin);
    end;
end
```

**Figure 6.** Procedures to update dependencies

nodes waiting on them that have a selected atom are disposed. All nodes waiting on them that have a selected ground negative literal are processed. The latter may lead to new nodes, which are processed by calling SLG_NEWCLAUSE.

Notice that both PosMin and NegMin are re-initialized to maxint. The previous values of PosMin and NegMin are obtained from those subgoals that are just completed, and thus should be discarded. However, the completion of those subgoals may create new nodes that are processed by calling SLG_NEWCLAUSE. The handling of the new nodes can introduce new subgoals that are pushed onto the stack. The PosMin and NegMin returned from the processing of those new nodes (and also from SLG_SUBGOAL($A$,PosMin,NegMin)) are used in UPDATE_SOLUTION to update the dependencies of the subgoal that leads to the creation of subgoal $A$.

If $\text{PosLink}(A) = \text{DFN}(A)$ and $\text{DFN}(A) \leq \text{NegLink}(A) < \text{maxint}$, $A$ and subgoals on top of $A$ may be involved in negative loops. Delaying is applied to all nodes that have a selected ground negative literal whose subgoal is $A$ or on top of $A$. The NegLink of $A$ and subgoals on top of $A$ are reset to maxint. Delaying creates some new nodes. The PosMin is reset to the DFN of the subgoal at the top of the stack and NegMin is reset to maxint before those newly created nodes are processed. When it finishes, $A$ and subgoals on top of $A$ are re-checked for completion.

### *4.3. Correctness of the Algorithm*

The correctness of SLG resolution, as proved in [6], is independent of the order in which transformations are applied. Our implementation uses a depth-first and tuple-at-a-time strategy to decide the order of transformations to be applied to the search forest represented by the global table of subgoals. For the correctness of the algorithm, it is sufficient to show that each transformation is implemented correctly and that when the evaluation of a query atom $A$ finishes, either $A$ is floundered, or $A$ and all relevant subgoals are completed. Theorem 2.1 guarantees that the program consisting of the answer clauses of $A$ and relevant subgoals preserve all three valued stable models of the original program, including the well-founded partial model.

All transformations except COMPLETION are implemented directly according to the definitions in Section 2.3. Although the decision of when to apply DELAYING is made based upon dependency information, the algorithm carries out DELAYING transformation following the definition.

The only exception is COMPLETION, which uses dependency information to derive subgoals that are completely evaluated. The following theorem shows that the implementation of COMPLETION is correct in the sense that all subgoals that are popped off are completely evaluated by Definition 6.

*Theorem 4.1. Let $P$ be a program and $Q$ be a query atom. Let $\mathcal{T}$ be the global table of subgoals and $\mathcal{S}$ be the global stack of subgoals. Then every subgoal $A$ in table $\mathcal{T}$ that is not on stack $\mathcal{S}$ is completed.*

PROOF. We prove by induction on the number of times subgoals are popped off the stack. The theorem holds initially since both $\mathcal{T}$ and $\mathcal{S}$ contains only the initial subgoal $Q$.

Subgoals are popped off the stack in procedure SLG_COMPLETE($A$,PosMin,NegMin), where $A$ is a subgoal, provided that $\text{PosLink}(A) = \text{DFN}(A)$ and $\text{NegLink}(A) = \text{maxint}$. Let $S_A$ be the set of subgoals from the top of stack $\mathcal{S}$ up to and including $A$. We show that $S_A$ are completely evaluated according to Definition 6.

```
procedure SLG_COMPLETE(A,PosMin,NegMin):
begin
    PosLink(A) := min(PosLink(A),PosMin);
    NegLink(A) := min(NegLink(A),NegMin);
    if PosLink(A)=DFN(A) and NegLink(A)=maxint then begin
        pop subgoals off stack S until A is popped;
        let S_A be the list of all popped subgoals;
        let L be the empty list;
        for each subgoal B ∈ S_A do begin
            Negs := Negs(B);
            Comp(B) := true; Poss(B) := []; Negs(B) := [];
            for each (A',G) ∈ Negs do begin
                if Anss(B)={} then begin
                    let G' be G with ~B deleted;
                    insert (A',G') into L;
                else if B :- | ∉ Anss(B) then begin
                    let G' be G with ~B delayed;
                    insert (A',G') into L;
                end;
            end;
            PosMin := maxint; NegMin := maxint;
            for each (A',G') in L do
                SLG_NEWCLAUSE(A',G',PosMin,NegMin);
    end else if PosLink(A)=DFN(A) and NegLink(A)≥DFN(A) then begin
        let S_A be the sequence of all subgoals from the top of S to A;
        let L be the empty list;
        for each subgoal B in S_A do begin
            for each (A',G) ∈ Negs(B) do begin
                let G' be G with the selected negative literal delayed;
                insert (A',G') into L;
            end;
            NegLink(B) := maxint; Negs(B) := [];
        end;
        PosMin := DFN(A'), where A' is at the top of stack S;
        NegMin := maxint;
        for each (A',G') in L do
            SLG_NEWCLAUSE(A',G',PosMin,NegMin);
        for each subgoal B in S_A do
            SLG_COMPLETE(B,PosMin,NegMin);
    end;
end
```

**Figure 7.** Procedure to complete a subgoal

First, for every subgoal $B \in S_A$, there is currently no new node that needs to be processed for $B$ and its relevant subgoals. The reason is that SLG_COMPLETE is called in SLG_SUBGOAL after all the child nodes of the root of the tree for $A$ have been fully processed, including all other new nodes created during the processing. Since $A$ is the first subgoal created among all subgoals in $S_A$, all new nodes that are created during the evaluation of $A$ have been processed when SLG_COMPLETE is called for $A$.

Second, let $G$ be any non-root node $v$ in the tree for a subgoal $B \in S_A$ and let $L$ be the selected literal of $G$.

- If $L$ is a non-ground negative literal, then computation must have been aborted, a contradiction.
- If $L$ is a ground negative literal of the form $\sim B'$, there are several cases:
  - If $B'$ is not in the table $\mathcal{T}$, then $B'$ is a new subgoal. SLG_SUBGOAL($B'$,BPosMin,BNegMin) is called. If $B'$ is not completed when SLG_SUBGOAL returns, NegLink($B$) is updated whose new value must be less than maxint, and so is the NegLink of $A$, a contradiction. If $B'$ is completed when SLG_SUBGOAL returns, $G$ must have been disposed when $B'$ is completed and $L$ either fails, succeeds, or is delayed.
  - If $B'$ is in the table $\mathcal{T}$ and is completed, then $G$ must have been disposed and $L$ either fails, succeeds, or is delayed.
  - If $B'$ is in the table $\mathcal{T}$ and is not completed, NegLink($B$) is updated whose new value must be less than maxint, and so is the NegLink of $A$, a contradiction.
- If $L$ is an atom, say $B'$, then there is a positive edge from $B$ to $B'$. There are several cases:
  - If $B'$ is a subgoal in $\mathcal{T}$, but not on stack, then $B$ is completed by inductive hypothesis. Neither the PosLink nor the NegLink of $A$ is updated in this case. All answers of $B'$ are returned to $G$.
  - If $B'$ is on stack $\mathcal{S}$, but not in $S_A$, then DFN($B'$) $<$ DFN($A$). Since there is a positive edge from $B$ to $B'$, the PosLink of $B$ must be less than DFN($A$) and so is the PosLink of $A$, a contradiction;
  - Otherwise, $B'$ must be on stack and in $S_A$. Since every new answer is returned immediately to all waiting nodes, and all existing answers are returned to a newly created node with a selected atom, all answers of $B'$ must have been returned to $G$.

By Definition 6, $S_A$ are completely evaluated. By COMPLETION transformation, subgoals in $S_A$ are popped off the stack and are marked as completed. □

In summary, every transformation in SLG resolution is implemented correctly by our algorithm. Let $P$ be a program and $A$ be the initial query atom. When SLG_SUBGOAL($A$,PosMin,NegMin) returns, the stack must be empty. This is because $A$ has the least depth-first number. By Theorem 4.1, $A$ and all relevant subgoals are completely evaluated by Definition 6. Therefore a final search forest has been constructed for $A$, all subgoals of which are completely evaluated. The correctness of the algorithm is then established by Theorem 2.1.

## 5. DISCUSSION

This section compares with related work and presents some performance measurements of two implementations of SLG resolution.

### *5.1. Related Work*

The framework of tabulated resolution for well-founded semantics by Bol and Degerstedt [3] defines a search space for query evaluation, which is similar to SLG resolution [6]. One interesting aspect of the approach in [3] is that non-ground negative literals are also returned as part of answers. This allows a more flexible handling of some queries that would be floundered in SLG resolution.

The bottom-up techniques presented in [10, 11, 13, 15] evaluate queries according to the alternating fixpoint [28] or the smallest three valued stable model [4, 17] in a more direct manner. The magic sets technique in [10, 11] may make too many magic facts true, and thus evaluate subgoals that are irrelevant. The improvement proposed by Morishita [13] alleviates this problem, but still generates many irrelevant magic facts in the initial stages of computing the alternating fixpoint.

*Example 5.1.* The following program is from [19].

$$p(X) \text{ :- } t(X,Y,Z), {\sim}p(Y), {\sim}p(Z).$$
$$p(X) \text{ :- } p0(X).$$

For query $p(a)$, the corresponding magic program is:

$$mp(a).$$
$$mp(Y) \text{ :- } mp(X), t(X,Y,Z).$$
$$mp(Z) \text{ :- } mp(X), t(X,Y,Z), {\sim}p(Y).$$
$$p(X) \text{ :- } mp(X), t(X,Y,Z), {\sim}p(Y), {\sim}p(Z).$$
$$p(X) \text{ :- } mp(X), p0(X).$$

This program is in fact an example where the well-founded semantics of the magic program does not agree with that of the original program, assuming the following facts for base predicates:

$$p0(c2).$$
$$t(a,a,b1). \quad t(b1,c1,b2). \quad t(b2,c2,b3). \quad t(bn,cn,cn+1).$$

Morishita's method [13] uses a slight variant of the alternating fixpoint. The early stages of the computation still generates many magic facts that are not relevant. For example, both the first positive overestimate and the second positive underestimate contain the following magic tuples:

$$mp(a). \quad mp(b1). \ldots mp(bn). \quad mp(c1). \ldots mp(cn+1).$$

Our implementation of SLG resolution generates only subgoals (or magic tuples) that are relevant, namely $p(a), p(b1), p(c1), p(b2), p(c2)$.

Ross first used subgoal dependencies in query evaluation with modularly stratified programs [21]. Facts representing transitive dependencies among subgoals are

computed explicitly. However, techniques for efficient maintenance and computation of subgoal dependencies were not explored.

The work most closely related to ours is the Ordered Search technique for bottom-up evaluation of left-to-right modularly stratified programs by Ramakrishnan *et al* [18]. An extension of Ordered Search, called *well-founded ordered search*, was recently proposed by Stuckey and Sudarshan [23]. The idea of Ordered Search is to simulate the subgoal dependencies induced by top-down evaluation. There are three interesting differences between (well-founded) ordered search and our implementation.

First, Ordered Search generates all answers of the first subgoal in the body of a clause before trying to solve the second subgoal in the body. We, however, follow closely the tuple-at-a-time computation of Prolog. As soon as an answer of the first subgoal in the body of a clause is generated, our implementation continues with the next subgoal in the body of a clause. This allows fast generation of the first answer for a query. In the case of a ground negative subgoal $\sim A$, as soon as a definitely true answer for $A$ is derived, $\sim A$ can fail and subgoals that are created during the evaluation of $A$ can be discarded under certain conditions (even if they are not fully evaluated). An additional benefit is the integration of Prolog with effective query evaluation. This objective has been achieved in XSB, where Prolog execution and SLG resolution are tightly interconnected. From the users' point of view, ordinary Prolog programs can be executed using SLG resolution with just a few declarations.

Second, Ordered Search maintains a topological order among all subgoals that have been expanded using a sequence of so-called *ContextNode*s. The topological order is based upon the dependency graph of subgoals. Each ContextNode may contain more than one subgoal when there are mutual dependencies among subgoals. A ContextNode is marked if some of its subgoals are marked, and subgoals are marked if their trees have been created and expanded. Each unmarked ContextNode contains a single subgoal whose tree has not yet been created. By re-arranging the sequence of ContextNodes at run time, strongly connected components in the dependency graphs can be identified.

In contrast, the stack of subgoals in SLG resolution behaves like the local stack of subgoals in Prolog. New subgoals are simply pushed onto the stack as they are encountered. There is no re-ordering of subgoals on the stack at run time. This may, however, cause unnecessary delaying and evaluation of some irrelevant subgoals, even when programs are stratified.

*Example 5.2.* Suppose that a query $m$ is evaluated with respect to the following program:

$m$ :- $c, \sim a, e$.
$c$ :- $b$.
$c$.
$b$ :- $c, d$.
$a$ :- $\sim b$.

Figure 1(i) shows the stack after the edge from $b$ to $c$ is traversed. The computation returns to subgoal $c$ and derives an answer using the second clause of $c$. The answer is returned to every waiting node, including the node in the tree for subgoal $m$. This leads to a new subgoal $a$. Figure 1(ii) shows the stack after

the negative from $a$ to $b$ is processed. The NegLink of $a$ is updated to 2, which is propagated to $b$ and $c$ through NegMin, creating a condition of a potential negative loop (even though the program is stratified). The negative subgoals $\sim a$ and $\sim b$ are delayed, leading to a new subgoal $e$ that is irrelevant to $m$ since $\sim a$ is false in the well-founded semantics.

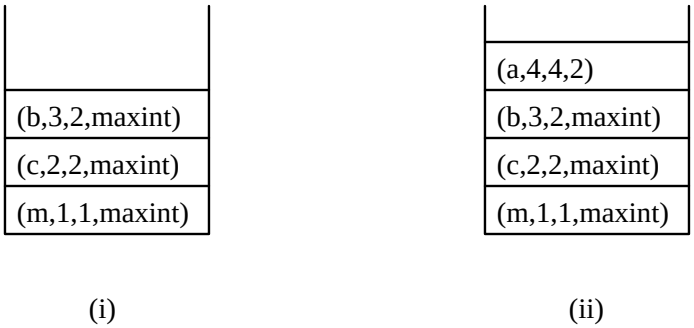


**Figure 1.** Stacks of subgoals indicating unnecessary delaying

The tradeoff is between maintaining precise dependencies through run-time reordering of subgoals on the stack and risking the evaluation of irrelevant subgoals. Which approach is more efficient in practice remains to be determined.

The third difference lies in the handling of negative loops. Well-founded ordered search uses the alternating fixpoint technique for subgoals involved in negative loops by calculating possibly true or false facts. Our implementation delays all selected ground negative literals possibly involved in negative loops. The negative edges are eliminated and the negative dependencies are reset. Delayed literals are simplified away later when their truth or falsity is known, but there is no redundant inference.

*Example 5.3.* The following program is from Example 4.1 in [24]:

$r(X)$ :- $\sim s(X)$.
$s(X)$ :- $q(X,Y), \sim r(Y), t(Y)$.
$q(X,a)$ :- $\sim r(X)$.

To handle negative loops, well-founded ordered search introduces predicates for computing true or undefined facts. The *Undef Magic* rewriting in [24, 23] produces the program below:

$r(X)$ :- $query(r(X)), done(s(X), \sim un(s(X))$.
$s(X)$ :- $query(s(X)), q(X,Y), done(r(Y)), \sim un(r(Y)), t(Y)$.
$q(X,a)$ :- $query(q(X,a)), done(r(X)), \sim un(r(X))$.
$un(r(X))$ :- $query(r(X)), un(\sim s(X))$.
$un(s(X))$ :- $query(s(X)), un(q(X,Y)), un(\sim r(Y)), un(t(Y))$.
$un(q(X,a))$ :- $query(q(X,a)), un(\sim r(X))$.
$un(r(X))$ :- $r(X)$.
$un(s(X))$ :- $s(X)$.
$un(q(X,a))$ :- $q(X,a)$.
$un(\sim r(X))$ :- $done(r(X)), \sim r(X)$.

$un(\sim s(X))$ :- $done(s(X)), \sim s(X)$.
$un(\sim q(X,a))$ :- $done(q(X,a)), \sim q(X,a)$.
$query^{\sim}(s(X))$ :- $query(r(X))$.
$query(q(X,Y))$ :- $query(s(X))$.
$query^{\sim}(r(Y))$ :- $query(s(X)), un(q(X,Y))$.
$query(t(Y))$ :- $query(s(X)), un(q(X,Y)), un(\sim r(Y))$.
$query^{\sim}(r(X))$ :- $query(q(X,a))$.

The systematic duplication of true facts in *un* causes redundant computation and extra space requirements for storing intermediate relations. The duplication is avoided in our implementation due to a uniform representation of answer clauses, which include both definitely true answers and possibly true answers that have delayed literals.

In the evaluation of query $r(a)$, there is a negative loop, involving $r(a)$, $s(a)$, and $q(a,Y)$. In well-founded ordered search, undefined facts are introduced: $un(\sim s(a))$ and $un(\sim r(a))$. This allows the computation to proceed and evaluate $t(a)$. The evaluation of $t(a)$ is completed and produces no answers. Well-founded ordered search returns to the ContextNode to evaluate the negative loop of $r(a)$, $s(a)$, and $q(a,Y)$, and starts alternating fixpoint computation for the negative loop, even though the negative loop has been broken since $s(a)$ fails. According to [24], the following sequence of actions is invoked:

- Add $done(s(a))$ (since a fixpoint has been reached and $un(s(a))$ is not present);
- Delete un-facts $un(q(a,a))$ and $un(r(a))$ (to begin the next stage of fixpoint computation);
- Fixpoint computation using the relevant rules in the magic program, which derives $un(q(a,a))$, $r(a)$ and $un(r(a))$;
- Add $done(r(a))$ (since $r(a)$ is now present);
- Remove $un(\sim r(a))$ (since $r(a)$ is now present);
- Delete un-facts $un(q(a,a))$ (to begin the next stage of fixpoint computation);
- Fixpoint computation again, producing no new facts. Thus the ContextNode for the negative loop is removed and $done(q(a,Y))$ is added.

Notice that $un(q(a,a))$ and $un(r(a))$ are deleted and then re-derived.

Our implementation delays $\sim s(a)$ and $\sim r(a)$, which is similar to adding undefined facts of $un(\sim s(a))$ and $un(\sim r(a))$. However, subgoals $r(a)$ and $q(a,Y)$ are both completely evaluated with conditional answers:

$r(a)$ :- $\sim s(a)$.
$q(a,a)$ :- $\sim r(a)$.

The subgoal $s(a)$ is completely evaluated with no answers since $t(a)$ fails. The failure of $s(a)$ is used to simplify the conditional answer for $r(a)$, and in turn, the success of $r(a)$ is used to delete the conditional answer for $q(a,Y)$. Two aspects should be noted. First, the derivation of conditional answers is not repeated. Second, the simplification of delayed literals is carried out only on conditional answers, which is much more efficient than a fixpoint computation using the corresponding clauses in the original magic program. Repeated derivation due to over-estimating the truth or undefinedness of subgoals is avoided.

It should be mentioned that repeated computation can occur due to the fact that variant checking is used for identifying duplicate subgoals. It is possible that both $p(X,Y)$ and $p(a,Y)$ are evaluated. Clearly all answers of $p(a,Y)$ are answers of $p(X,Y)$ (unless Prolog builtin predicates like $var/1$ are used in the definition of $p/2$). Subsumption checking of subgoals is needed to avoid such repetition.

## *5.2. Performance Measurements*

There are two freely available implementations that make use of the algorithms in this paper. The SLG system, which is a meta interpreter written in Quintus Prolog, implements the algorithms fully. Another, the SLG-WAM of XSB compiles a restriction of SLG for left-to-right modularly stratified programs[25]. (The SLG-WAM is currently being extended to evaluate the full SLG resolution). To get a rough idea how the meta interpreter and XSB perform, we took the benchmark programs reported in [13] together with their timing information, and then ran them using the meta interpreter and XSB. However, it should be pointed out that a systematic study of benchmark that include negation has to be conducted before a clear picture of the relative performance of the various systems can be obtained (for definite programs, systematic experiments have been reported in [26]).

The following experiments are taken from [13]. The intensional database contains only one rule:

$$win(X) \text{ :- } move(X,Y), \sim win(Y).$$

Three different relations for *move* are used, one containing an acyclic linear list: (1,2), ..., (N-1,N), another containing a complete binary tree of height $H$ (with $2^{H+1}-1$ tuples), and the other containing a cyclic linear list: (1,2), ..., (N-1,N), (N,1). Execution times were provided in [13] for query $win(1)$ in Glue-Nail's implementations of Ross's method for modularly stratified programs [19] and Morishita's alternating fixpoint tailored to magic programs [13].

We ran the meta interpreter implementation of SLG resolution on these programs using Quintus Prolog 3.1 on a Decstation 3100 (Ultrix V4.2A (Rev. 47)). The timing information in each experiment was obtained using the builtin predicate statistics/2 in one run. For the two modularly stratified programs, we re-ran the SLG meta interpreter against XSB on a SPARCstation 2. The average of 100 iterations was taken in comparing the meta interpreter to the emulator.

The following tables show the execution times (in seconds) of our meta interpreter in comparison with the timing information from [13]. The numbers for Morishita's implementation were taken from a DEC 5000 [14], a slightly faster machine than the DECstation 3100. In addition, meta interpreter times are also shown *normalized* to XSB's SLG evaluation. negation.

The results seem to indicate that our meta interpreter is competitive with Morishita's implementation, and that the XSB system is an order of magnitude or more faster than the meta interpreter. Morishita's implementation performs better for cyclic linear lists than for acyclic linear lists. This is due to the fact that all *win* facts are undefined in the cyclic case and the fixpoint is immediately reached [13]. On the other hand, the execution times of the SLG meta interpreter are comparable in both cases of linear lists. The delaying in the cyclic case makes the meta interpreter slightly slower than in the acyclic case.

| N | 8 | 16 | 32 | 64 | 128 | 256 |
|---|---|---|---|---|---|---|
| SLG | 0.050 | 0.100 | 0.233 | 0.467 | 0.933 | 2.000 |
| Morishita | 0.199 | 0.715 | 2.621 | 10.18 | 40.79 | 161.9 |
| Ross | 0.145 | 0.309 | 0.738 | 2.05 | 6.56 | 23.7 |

**Table 1.** Timing for acyclic linear lists

| H | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|
| SLG | 0.934 | 1.934 | 4.084 | 13.18 | 28.02 | 63.45 |
| Morishita | 1.11 | 2.64 | 5.24 | 12.5 | 25.0 | 59.6 |
| Ross | 1.62 | 4.12 | 10.86 | 33.6 | 111.0 | 398.4 |

**Table 2.** Timing for complete binary trees

| N | 8 | 16 | 32 | 64 | 128 | 256 |
|---|---|---|---|---|---|---|
| SLG | 0.067 | 0.134 | 0.283 | 0.600 | 1.233 | 2.550 |
| Morishita | 0.055 | 0.094 | 0.180 | 0.348 | 0.691 | 1.391 |

**Table 3.** Timing for cyclic linear lists

| Length | 8 | 16 | 32 | 64 | 128 | 256 |
|---|---|---|---|---|---|---|
| SLG Interp. | 30 | 30 | 33 | 32 | 29 | 29 |
| XSB | 1 | 1 | 1 | 1 | 1 | 1 |

**Table 4.** SLG engine and interpreter for acyclic linear lists

| Height | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|
| SLG Interp. | 28 | 27 | 30 | 32 | 53 | 60 |
| XSB | 1 | 1 | 1 | 1 | 1 | 1 |

**Table 5.** SLG engine and interpreter for complete binary trees

Further benchmarks of XSB for these programs show linear performance as the database size is increased through 32k for linear lists, and through 64k for trees. In summary these preliminary benchmark results seem to indicate that XSB outperforms prototypes of deductive databases in most cases, and can be significantly faster. XSB also provides an alternate form of negation for SLG evaluation which can further optimize these programs.

### *5.3. Existential Negation in XSB*

SLG evaluation as defined in this paper will not cause the exponential behavior that can be observed in some other top-down approaches [7], because it fully evaluates all subgoals even when they are created as a result of a call to a negative subgoal. This method of evaluation is inefficient for the *win/1* example over the binary tree. To see this, consider the calls made by SLDNF for the query *win(1)* over a binary tree with 31 nodes. The calls are represented as circled nodes in Figure 2. Because SLDNF checks only for the existence of a solution for a negative subgoal, only 13 out of 31 possible subgoals are evaluated by SLDNF, and in general the execution of *win(1)* over a binary tree grows proportionally to $\sqrt{2}^n$ in SLDNF rather than to $2^n$. [1]

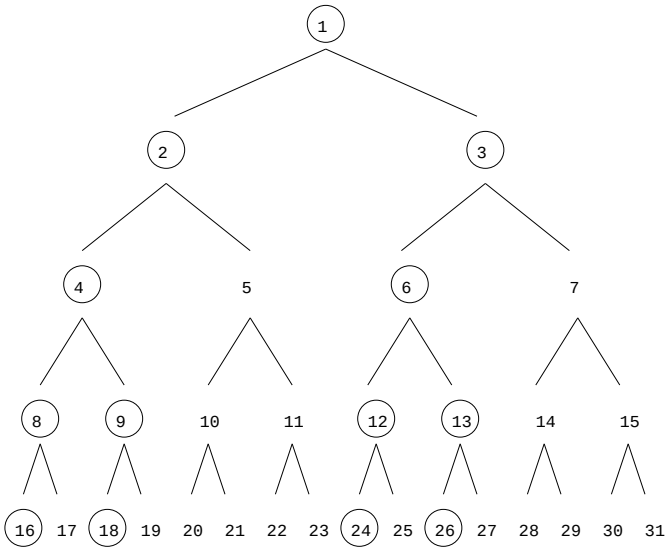


**Figure 2.** Calls to `win/1` over a binary tree

Version 1.4 of XSB allows three different ways of executing *win/1*. The first uses pure SLG resolution in which all subgoals are fully evaluated. This method is used in the comparison with the SLG meta interpreter in Table 4 and Table 5. The second uses SLDNF resolution. *Existential negation* is the third alternative of XSB, which combines some of the search strategy of SLDNF resolution with SLG resolution. In existential negation, when a definitely true answer is derived for $A$, the

[1] The exact formula is $G(n) = 2^{\lfloor \frac{n}{2} \rfloor + 2} - 3 + 2(\frac{n}{2} - \lfloor \frac{n}{2} \rfloor)$.

corresponding ground negative subgoal $\sim A$ fails. Furthermore, subgoals that are created during the evaluation of $A$ can be discarded without being fully evaluated under certain conditions without losing termination and correctness properties of SLG resolution. The tables below show normalizations of the execution times of the SLG meta interpreter and the first two methods of XSB to that of XSB with existential negation for the two benchmark programs that are modularly stratified.

| Length | 8 | 16 | 32 | 64 | 128 | 256 |
|---|---|---|---|---|---|---|
| SLG Interp. | 30 | 30 | 33 | 32 | 29 | 29 |
| XSB / No E-Neg | .99 | .99 | 1 | .99 | 1 | .97 |
| XSB / SLDNF | .67 | .21 | .22 | .22 | .24 | .26 |
| XSB / E-Neg | 1 | 1 | 1 | 1 | 1 | 1 |

**Table 6.** Comparisons of SLG implementations for acyclic linear lists

| Height | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|
| SLG Interp. | 123 | 116 | 229 | 261 | 812 | 906 |
| XSB / No E-Neg | 4.5 | 4.25 | 7.6 | 8.2 | 15.4 | 15.7 |
| XSB / SLDNF | .3 | .24 | .22 | .24 | .24 | .23 |
| XSB / E-Neg | 1 | 1 | 1 | 1 | 1 | 1 |

**Table 7.** Comparisons of SLG implementations for complete binary trees

## 6. CONCLUSION

We have presented efficient techniques for implementing SLG resolution [6], which is a transformational framework for computation of queries with respect to the well-founded semantics. We firmly believe that SLG resolution will have an important impact on the theory and practice of logic-based computational systems. Its termination properties on stratified Datalog programs make it a good strategy for deductive database query processing; its ability to be integrated seamlessly with Prolog evaluation makes it a good logic programming strategy, and its polynomial data complexity for handling nonstratified Datalog programs makes it a good strategy for nonmonotonic knowledge representation problems.

Implementation techniques developed in this paper not only bring the declarative semantics of logic programs to Prolog programmers and other users, but also are applicable to problems that involve various extensions of logic programs, including constructive negation and constraint logic programming.

## ACKNOWLEDGMENT

We thank S. Sudarshan for discussions on the (well-founded) ordered search techniques and S. Dawson and K. Sagonas for their help on the time complexity of

SLDNF resolution for query $win(1)$ in the case of a complete binary tree. We thank anonymous referees for their careful reading and helpful comments.

## REFERENCES


1. Apt, K.R., Blair, H., AND Walker, A. Towards a theory of declarative knowledge. In Minker, J., editor, *Foundations of Deductive Databases and Logic Programming*, pages 89–148. Morgan Kaufmann Publishers, Los Altos, CA, 1988.
2. Bidoit, N. AND Legay, P. Well!: An evaluation procedure for all logic programs. In *Intl. Conference on Database Theory*, pages 335–348, 1990.
3. Bol, R. AND Degerstedt, L. Tabulated resolution for well founded semantics. In *Intl. Logic Programming Symposium*, October 1993.
4. Bonnier, S., Nilsson, U., AND Näslund, T. A simple fixed point characterization of three valued stable model semantics. *Information Processing Letters*, 40(2):73–78, 1991.
5. Chen, W. AND Warren, D.S. A goal-oriented approach to computing well founded semantics. *Journal of Logic Programming*, 17, 1993.
6. Chen, W. AND Warren, D.S. Query evaluation under the well founded semantics. In *The Twelfth ACM Symposium on Principles of Database Systems*, 1993.
7. Chen, W. AND Warren, D.S. Towards effective evaluation of general logic programs. Technical Report 93-CSE-11, Department of Computer Science and Engineering, Southern Methodist University, 1993.
8. Chen, W. AND Warren, D.S. *The SLG System*, August, 1993. available by anonymous FTP from seas.smu.edu or cs.sunysb.edu.
9. Dietrich, S.W. AND Warren, D.S. Extension tables: Memo relations in logic programming. Technical Report 86/18, Department of Computer Science, SUNY at Stony Brook, 1986.
10. Kemp, David B., Stuckey, Peter J., AND Srivastava, Divesh. Magic sets and bottom-up evaluation of well-founded models. In *Intl. Logic Programming Symposium*, pages 337–351, 1991.
11. Kemp, David B., Stuckey, Peter J., AND Srivastava, Divesh. Query restricted bottom-up evaluation of normal logic programs. In *Joint Intl. Conference and Symposium on Logic Programming*, pages 288–302, 1992.
12. Lloyd, J.W. *Foundations of Logic Programming*. Springer-Verlag, New York, second edition, 1987.
13. Morishita, S. An alternating fixpoint tailored to magic programs. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, 1993.
14. Morishita, S. Personal communication, 1993.
15. Nilsson, U. Goal-directed bottom-up evaluation of normal logic programs. Manuscript, 1993.
16. Przymusinski, T.C. Every logic program has a natural stratification and an iterated least fixed point model. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 11–21, 1989.
17. Przymusinski, T.C. The well-founded semantics coincides with the three-valued stable semantics. *Fundamenta Informaticae*, 13:445–463, 1990.
18. Ramakrishnan, R., Srivastava, D., AND Sudarshan, S. Controlling the search in bottom-up evaluation. In *Joint Intl. Conference and Symposium on Logic Programming*, pages 273–287, 1992.
19. Ross, K.A. Modular stratification and magic sets for datalog programs with negation. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 161–171, 1990.

20. Ross, K.A. A procedural semantics for well-founded negation in logic programs. *Journal of Logic Programming*, 13(1):1–22, 1992.
21. Ross, K.A. *The Semantics of Deductive Databases.* PhD thesis, Department of Computer Science, Stanford University, August 1991.
22. Sagonas, K., Swift, T., AND Warren, D.S. XSB as an efficient deductive database engine. In *ACM SIGMOD Conference on Management of Data*, pages 442–453, 1994.
23. Stuckey, P. AND Sudarshan, S. Well-founded ordered search. In *Proceedings of the 13th Conference on Foundations of Software Technology and Theoretical Computer Science*, 1993. LNCS 761.
24. Stuckey, P. AND Sudarshan, S. Well-founded ordered search. Manuscript, May 12 1993.
25. Swift, T. AND Warren, D. S. An abstract machine for SLG resolution. In *Intl. Logic Programming Symposium*, 1994.
26. Swift, T. AND Warren, D. S. Analysis of sequential SLG evaluation. In *Intl. Logic Programming Symposium*, 1994.
27. Tamaki, H. AND Sato, T. Old resolution with tabulation. In *Intl. Conference on Logic Programming*, pages 84–98, 1986.
28. van Gelder, A. The alternating fixpoint of logic programs with negation (extended abstract). In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 1–10, 1989.
29. Van Gelder, A., Ross, K.A., AND Schlipf, J.S. The well-founded semantics for general logic programs. *Journal of ACM*, 38(3), July 1991.
30. Vieille, L. A database-complete proof procedure based upon sld-resolution. In *Intl. Conference on Logic Programming*, 1987.