

A Formal Framework to Model Scheduling in Tabled Evaluations

Juliana Freire* Terrance Swift† David S. Warren‡

Abstract

Tabled evaluation ensures termination for a large class of programs by keeping track of which subgoals have been called. Given several variant subgoals in an evaluation, only the first one encountered will use program clause resolution; the rest will resolve with the answers generated by the first subgoal. This use of answer resolution prevents infinite looping which sometimes happens in SLD. Because answers that are produced in one path of the computation may be consumed, asynchronously, in others, tabling systems face an important scheduling choice not present in traditional top-down evaluation: when to schedule answer resolution.

Various scheduling strategies have been developed and implemented in the XSB tabling system. Experiments have shown that the efficiency of an evaluation is highly sensitive to the order in which operations are performed. However, the lack of a formal structure made it difficult to define these strategies more precisely, prove their correctness and compare different strategies. Here, we address this problem by proposing SLG_{sched} , an extension of SLG that provides a formal framework to model scheduling strategies

1 Introduction

Applications of Tabled Logic Programming are becoming common. Many abstract interpreters use tabling (or magic) to determine properties of programs [?, ?, ?]. Tabling is also used for model checking [?], for natural language analysis [?, ?], and for medical diagnosis [?]. Several reasons account for this profusion of applications, among them the fact that tabled logic programs have a clear declarative semantics in the Well-Founded Semantics, several well-defined operational semantics, and high-quality implementations such as the XSB system [?] among others

Experience has indicated that tabled evaluations can be highly sensitive to the order in which operations are performed, or the *scheduling strategy* for an evaluation. Previous work [?, ?, ?] has shown that the scheduling strategy used can radically affect memory usage, execution time, or disk access patterns (this work is briefly summarized in Section 4). As a result, it seems likely that tabling systems of the future will evaluate large programs using a mixture of scheduling strategies [?] determined by analysis of a program's requirements.

Questions about scheduling also arise when comparisons are made between implementations. While [?] has demonstrated that for definite programs, relevant properties of magic-style evaluations can be captured by a special scheduling strategy for tabling, the question of comparing tabled evaluations with magic-based evaluations that include negation, such as well-founded ordered search [?] or the method of [?] remains open. Similar questions about scheduling occur when tabling is combined with parallelism [?, ?].

Work on both of these questions — on mixing scheduling strategies and on comparing evaluation methods — has been made difficult by the lack of a formal basis for scheduling in tabled evaluations. Various formulations either rely implicitly on a given scheduling strategy, such as OLDT [?] or WFOS [?], or do not formulate scheduling at all as in SLG [?].

*Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974. Email: juliana@research.bell-labs.com

†Department of Computer Science, University of Maryland, College Park, MD. Email: tswift@cs.umd.edu

‡Department of Computer Science, SUNY at Stony Brook, Stony Brook, NY 11794-4400. Email: warren@cs.sunysb.edu

In this paper we present SLG_{sched} , a framework for modeling scheduling in SLG evaluations. As will be seen, SLG_{sched} clarifies interrelations between tabling operations, making it easier to prove the correctness both of the scheduling strategies themselves and of low-level optimizations based on these scheduling strategies.

The results of this paper are as follows:

- We formally define the framework of Scheduled SLG Evaluations. These evaluations augment SLG forests with a *scheduling sequence*, and parameterize evaluations using sequence combinators. We then show that Scheduled SLG Evaluations using sequence combinators have the same generality as SLG evaluations.
- We show that by altering the sequence combinator parameter, four evaluation strategies can be derived: Single Stack Scheduling, Batched Scheduling, Local Scheduling, and Breadth-First Scheduling. Each of these evaluations has proven useful enough to have been implemented in the XSB system.
- We then use the SLG_{sched} formalism to show that the breadth-first strategy is equivalent to magic for definite programs.

The structure of the paper is as follows. Section 2 reviews SLG evaluation, which has been formally presented in [?]. In Section 3 we introduce SLG_{sched} , an extension of SLG that takes scheduling into account, and show how SLG_{sched} evaluations can be parameterized by their manipulation of the scheduling sequence. In Section 4 we give examples of how different scheduling strategies can be represented through parameterizing Scheduled SLG Evaluations. In this section, we also show how the formalization can be used to compare Breadth-First Scheduling to magic-style evaluations, and to compare the efficiency of local to non-local evaluations when answer subsumption is used.

2 SLG: A Brief Review of Terminology

SLG resolution (Linear resolution with Selection function for General logic programs) [?] is a tabled evaluation method that is sound and search space complete with respect to the well-founded partial model for all non-floundering queries. In this section we review certain definitions of SLG resolution from [?] that will be used in Section 3 to define SLG_{sched} . Because of space limitations, we cannot provide all definitions here, but refer the reader to [?] and to the full version of this paper.¹

Unless otherwise noted, throughout this document we restrict ourselves to non-floundering finite programs. We also assume, without loss of generality, that all negative subgoals are tabled, and that a left-to-right computation rule is used.

Preliminary Definitions

As preliminary terminology, subgoals are atoms. Two atoms are identical if they are variants of each other. Predicates can be annotated as either tabled or non-tabled, in which case SLD resolution is used.

An SLG evaluation is modeled by a sequence of *forests* of SLG trees. Figure 1, which will be discussed in detail below, illustrates the SLG forest for a query to a simple recursive program.

Definition 2.1 (SLG Forest) An SLG forest consists of a set of SLG trees. The root nodes of SLG trees have the following form:

$$Subgoal \leftarrow |Subgoal$$

¹For the convenience of referees, we provide definitions of SLG operations and of SLG evaluation in Appendix B.

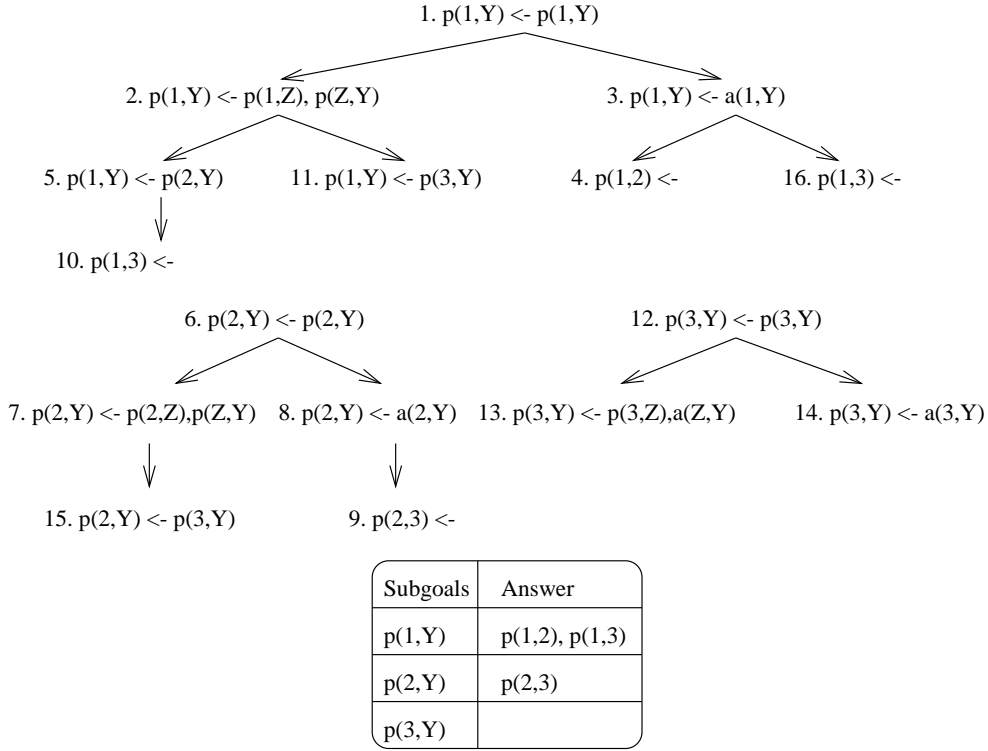


Figure 1: A finite forest for tabled evaluation

and the status of a root node may be set to *completed* when its corresponding tree is *completely evaluated* (Definition 2.4). Non-root nodes can have the form:

$$AnswerTemplate \leftarrow DelaySet | GoalList$$

where *AnswerTemplate* is an atom, *GoalList* is a sequence of literals, and *DelaySet* is a set of delayed literals. Non-root nodes can also be *failed*. Failed nodes are added as leaves of failed branches and are designated by the atom *fail*. \square

In non-root nodes, the *AnswerTemplate* captures variable bindings accumulated during the evaluation. *GoalList* is a sequence of literals that remain to be solved. *DelaySet* is a set of annotated positive or negative literals. If *S* is the selected literal for a non-root node *N*, *N* is called a *consuming node* of *S*.

Because we assume non-floundering programs, negative literals are always ground, and so will have no special form in a *DelaySet*. Positive literals require an extra annotation and have the form D_{Answer}^{Call} , where *D* is an atom whose truth value depends on the truth value of some answer *Answer* for the subgoal *Call*. If θ is a substitution, then $(D_{Answer}^{Call})\theta = (D\theta)_{Answer}^{Call}$. Intuitively, these annotations provide control information for a possible simplification of the delayed literal.

Definition 2.2 (Answer) A leaf node in an SLG tree is an *answer* if its goal list is empty. Given an answer $AnswerTemplate \leftarrow DelaySet$, if *DelaySet* is empty, the answer is said to be *unconditional*, otherwise, if *DelaySet* is not empty, the answer is said to be *conditional*. \square

Definition 2.3 (SLG Answer Resolution) Let \mathcal{F} be a forest. Let *G*, of the form $A \leftarrow D | L_1, \dots, L_n$, be a node in \mathcal{F} with selected positive literal L_i , for some i ($1 \leq i \leq n$). If *S* is a subgoal in \mathcal{F} that has an answer *Answer* of the form $A' \leftarrow D'$, whose variables have been standardized apart from *G*, we say that *G* is *SLG resolvable* with *Answer*. The

resolvent of G and $Answer$ on L_i has the form

$$(A \leftarrow D | L_1, \dots, L_{i-1}, L_{i+1}, \dots, L_n) \theta$$

if D' is empty, and

$$(A \leftarrow D, D' | L_1, \dots, L_{i-1}, L_{i+1}, \dots, L_n) \theta$$

otherwise, where $D' = S_{A'}^S$. □

If \mathcal{F} has a tree with root $S \leftarrow | S$, we say that S is contained in \mathcal{F} . In an SLG forest, no two SLG trees have the *same* root nodes, that is, their corresponding subgoals cannot be variants of each other. SLG uses the notion of a set of subgoals being completely evaluated in order to allow negative literals to succeed or fail.

Definition 2.4 (Completely Evaluated) Given an SLG forest, and a set $Subg$ of tabled subgoals, $Subg$ is *completely evaluated* iff at least one of the following conditions is satisfied for each subgoal $S \in Subg$:

1. S has an answer that is a variant of S ; or
2. For each node N in a tree with root S
 - The selected literal SL of N is completed, or
 - $SL \in Subg$ and no SLG operations SUBGOAL CALL, PROGRAM CLAUSE RESOLUTION, ANSWER CLAUSE RESOLUTION, NEGATIVE DELAY, or NEGATIVE RETURN (see Definition B.2) are applicable to N . □

If the tree for a subgoal S contains an unconditional answer that is a variant of S , S is said to *succeed*. If the tree for S is completed but contains no answers, S is said to be *failed*. After a subgoal is completely evaluated, answers that have positive literals in their bodies that are *unsupported* must be removed.

Definition 2.5 (Supported Answer) Let S be a subgoal and $Answer$ be an atom that occurs in the head of some answer of S . Then $Answer$ is supported by S if:

1. Either S does not have status completed; or
2. There exists an answer $Answer'$ of S that has $Answer$ in its head and for every positively delayed literal D_{Ans}^{Call} in $Answer'$, Ans is supported by $Call$. □

Definition 2.6 (Unsupported Answer) A set of answers \mathcal{UA} is unsupported if $\forall ua_i \in \mathcal{UA}$, where ua_i is of the form $A_i \leftarrow D_i$, the tree for ua_i has status completed and $\exists ua_j \in \mathcal{UA}$ such that $ua_j \in D_i$ (i.e., there exists a positive loop among the answers in \mathcal{UA}). □

SLG evaluations are modeled as a possibly transfinite sequence of forests. Any non-final SLG forest has a set of applicable SLG operations. New operations can become applicable in a forest if: a new tree is added; a new node is created; or the status of a root node is set to completed.

Example 2.1 Consider the following program that computes the transitive closure of a directed graph:

```
:- table p/2.
p(X,Y) :- p(X,Z), p(Z,Y).
p(X,Y) :- a(X,Y).
a(1,2). a(1,3). a(2,3).
```

As an illustration of the operations in Definition B.2 for definite programs, consider Figure 1 which represents forest for the query $:- p(1, Y)$ at the end of its evaluation. The first time a subgoal S is encountered during a tabled

evaluation, the SUBGOAL CALL operation creates a new tree with root $S \leftarrow S$.² Figure 1 contains trees rooted at nodes 1, 6 and 12 (we will also refer to these roots as *generator nodes*). PROGRAM CLAUSE RESOLUTION is then used to obtain the immediate children for each root node. Alternatively, if the forest does not contain S , no new tree is required for S . Processing of answers is analogous to subgoals: the first time an answer to a subgoal S is derived during an evaluation, it is returned to relevant consuming nodes of S via ANSWER CLAUSE RESOLUTION (for instance, nodes 5, 11 and 15 are produced by resolving answers against consuming nodes); any subsequently derived answers are not returned. In this manner, tabling prevents redundant subcomputations (including loops). As mentioned above, tabled resolution can be mixed with the program clause resolution of SLD. In this case, we refer to tabled predicates and non-tabled predicates depending on the form of resolution used for each. In an SLG tree, nodes that correspond to non-tabled predicates are termed *interior nodes*. In Figure 1, nodes 3, 8 and 14 are interior nodes.

In a tabled evaluation, groups of mutually dependent subgoals are called *strongly connected components* or SCCs. When all program and answer clause resolution has been performed for the subgoals in an SCC, the subgoals are termed *completely evaluated* (Definition 2.4). The notion of completion is necessary for evaluation of programs with negation, as well as being useful for space reclamation. In the forest depicted in Figure 1, there are three SCCs: $\{p(1, Y)\}$, $\{p(2, Y)\}$, $\{p(3, Y)\}$. The sets of subgoals $\{p(3, Y)\}$, $\{p(2, Y), p(3, Y)\}$, and $\{p(1, Y), p(2, Y), p(3, Y)\}$ are all completely evaluated — the evaluation has produced all possible answers for subgoals in these sets — and the COMPLETION operation can be applied to any of these sets of subgoals. \square

Correctness of SLG was shown in [?]. To restate this result, we briefly review some terminology. Let \mathcal{F} be a forest in an SLG evaluation of a program P and query Q . The partial interpretation of \mathcal{F} , $I(\mathcal{F})$, is a set of ground atoms constructed as follows. $A \in I(\mathcal{F})$ iff A is a ground instance of some answer in \mathcal{F} ; $not(A) \in I(\mathcal{F})$ iff A is a ground instance of some A' , and the SLG tree for A' is completed in \mathcal{F} but does not contain A as an instance of any answer. The following theorem states the correctness of SLG by relating the partial interpretations of final forests to the well-founded model of P restricted to the set of subgoals \mathcal{S} in \mathcal{F} ($\mathcal{M}_P|_{\mathcal{S}}$).

Theorem 2.1 ([?]) Let Q be a query to a normal program P . Then an SLG evaluation consisting of the operations SUBGOAL CALL, PROGRAM CLAUSE RESOLUTION, ANSWER CLAUSE RESOLUTION, COMPLETION, NEGATIVE DELAY, NEGATIVE RETURN, SIMPLIFICATION and ANSWER COMPLETION will reach a final forest \mathcal{F}_α in which a ground atom A is in $I(\mathcal{F}_\alpha)$ iff it is in $\mathcal{M}_P|_{\mathcal{S}}$, the well-founded model of P restricted to the set of subgoals in \mathcal{F} .

2.1 Adding Operational Features to SLG

In order to model operational features of SLG such as scheduling, it is useful have mechanisms that describe the dependencies of subgoals and the order of creation of nodes in a forest.

Definition 2.7 (Subgoal Dependency Graph) Let \mathcal{F} be an SLG forest. We say that a tabled subgoal S_1 *directly depends on* a tabled subgoal S_2 in \mathcal{F} iff S_1 and S_2 are *non-completed* and S_2 is the selected literal of some node in the tree for S_1 . The *subgoal dependency graph* of \mathcal{F} , $SDG(\mathcal{F})$, is a directed graph $\{V, E\}$ in which V is the set of root subgoals for non-completed trees in \mathcal{F} and $(S_i, S_j) \in E$ iff S_i directly depends on S_j . \square

We use the relation *depends on* to denote the transitive closure of the relation *directly depends on*. Since $SDG(\mathcal{F})$ is a directed graph, it can be partitioned into strongly connected components, or SCCs. An SCC is *independent* if no subgoal in that SCC depends on subgoals that belong to a different SCC.

We model the order of creation of nodes using *indexed forests*, in which each node (with the exception of failed nodes) is prepended by the index of the system in which the node was added — this number uniquely identifies each non-failed node in a forest. Using these numbers, we can describe the *calling environment* of a subgoal S as the consuming node of S whose index is least. The notion of a leader of an SCC combines the notions of indices and dependencies.

²For conciseness, in definite programs where no subgoals are delayed, root nodes are represented by *Subgoal* \leftarrow *Subgoal*, and non-root nodes are represented by *AnswerTemplate* \leftarrow *GoalList*.

Definition 2.8 (Leader of an SCC) A subgoal S is the leader of an SCC $Subg$, if S is the subgoal in $Subg$ with the smallest index. \square

Definition 2.9 (Completely Evaluated: Operational Formulation) Let $Subg$ be a set of subgoals in a given SLG forest \mathcal{F} . $Subg$ is *completely evaluated* if at least one of the following conditions is satisfied for each subgoal $S \in Subg$.

- S has an answer that is a variant of S ; or
- $Subg$ is an *independent SCC* (or contains a set of independent SCCs) and there are no SUBGOAL CALL, PROGRAM CLAUSE RESOLUTION, ANSWER CLAUSE RESOLUTION, NEGATIVE DELAY, or NEGATIVE RETURN, applicable to the nodes in the trees for S and for the other subgoals in $Subg$. \square

3 SLG_{sched}: Scheduling in SLG

An SLG_{sched} evaluation consists of a sequence of *systems*. Each system contains an indexed forest and a scheduling sequence. Scheduling sequences are sequences of scheduling tuples (Definition B.3). Each scheduling tuple uniquely represents an SLG_{sched} operation to be applied to a node in an indexed forest.³ For instance, a scheduling tuple $ACR(NodeNumber, AnswerNumber)$ would apply ANSWER CLAUSE RESOLUTION of the answer indexed by *AnswerNumber* against the selected atom of the node indexed by *NodeNumber*.

Given an SLG_{sched} system (\mathcal{F}_n, Seq_n) , a new SLG_{sched} system is produced as follows. The operation represented by the first scheduling tuple of Seq_n , op_0 , is applied to \mathcal{F}_n to produce a new forest \mathcal{F}_{n+1} . In addition, the definition of SLG_{sched} operations indicates how a new scheduling sequence Seq_{n+1} is produced — that is, which scheduling tuples should be added and/or deleted from Seq_n . Scheduling tuples are always selected from the *head* of a sequence, and are removed from the sequence after having been applied. A *sequence combinator* governs where in the scheduling sequence new operations are added.

Definition 3.1 (Sequence Combinator) Let \oplus be a function with signature $\oplus : Seq \times Seq \rightarrow Seq$. We call \oplus a *sequence combinator* iff given two sequences Seq_1 and Seq_2 , $Seq_1 \oplus Seq_2 = Seq_3$, and both Seq_1 and Seq_2 are subsequences of Seq_3 . \square

An SLG_{sched} evaluation is formulated as follows.

Definition 3.2 (SLG_{sched} Evaluation) Given a finite non-floundering program P , and sequence combinator \oplus , an SLG_{sched} evaluation \mathcal{E} for a tabled goal Q is a sequence of tuples $(\mathcal{F}_0, Seq_0), (\mathcal{F}_1, Seq_1), \dots, (\mathcal{F}_n, Seq_n)$, such that:

- \mathcal{F}_0 is the forest containing the initial query, θ . Q , and Seq_0 is the scheduling sequence containing the SLG operations applicable to θ . Q , $Seq_0 = SC(0)$ (i.e., the initial operation is to apply SUBGOAL CALL to subgoal Q).
- For each finite ordinal $n + 1$, \mathcal{F}_{n+1} and Seq_{n+1} are obtained from \mathcal{F}_n by applying the SLG_{sched} operation represented by the first element of Seq_n (see Definition A.1).

If $Seq_n = \emptyset$, no operation is applicable in \mathcal{F}_n , and (\mathcal{F}_n, Seq_n) is called a *final system* of \mathcal{E} . For each forest \mathcal{F}_i in the evaluation, we denote by $App_{\mathcal{F}_i}$ the set of SLG operations that are applicable in \mathcal{F}_i . \square

A sequence combinator is simply a way of merging two scheduling sequences. We say that a sequence combinator \oplus is *fair* if given a program P of bounded-term depth ([?]), every SLG_{sched} evaluation of a query Q to P using \oplus terminates. Intuitively, the reason for requiring fairness is to guarantee that for a finite query a specific strategy will not

³Each SLG operation has an SLG_{sched} counterpart, for example: ACR for ANSWER CLAUSE RESOLUTION, SC for SUBGOAL CALL, PCR for PROGRAM CLAUSE RESOLUTION, Compl for COMPLETION.

go into an infinite loop by, for instance, repeatedly selecting and pushing back the same operation into the scheduling sequence. Using the framework of SLG_{sched} , scheduling strategies are captured by defining a sequence combinator \oplus in an appropriate manner and proving it to be fair. The generality of this approach is assured by the following theorem.

Theorem 3.1 *Let P be a program with bounded-term depth, Q an atomic query to P . Then an SLG evaluation $\mathcal{E} = \{\mathcal{F}_0, \mathcal{F}_1, \dots, \mathcal{F}_n\}$ exists iff there is an SLG_{sched} evaluation*

$$\mathcal{E}' = \{(\mathcal{F}_{i_0}, Seq_0), (\mathcal{F}_{i_1}, Seq_1), \dots, (\mathcal{F}_{i_m}, Seq_m)\}$$

with a fair sequence combinator, such that $\forall k \in \{0, \dots, n\}, \exists k' \in \{0, \dots, m\}, i_{k'} \geq k$ such that $\mathcal{F}_k = \mathcal{F}_{i_{k'}}$.

Proof: For the convenience of the referees, the proof is given in Appendix B.2. ◇

4 Defining Scheduling Strategies

We now illustrate how the framework described in Section 3 can be used, by explaining how Single Stack Scheduling, Batched Scheduling, Local Scheduling and Breadth-First Scheduling [?, ?, ?] are formulated under SLG_{sched} . We note that each of these scheduling strategies was formulated for implementation purposes, and often were driven by low-level design decisions ⁴.

4.1 Single Stack Scheduling

Single Stack Scheduling was the first scheduling strategy implemented for SLG. The key idea of this strategy is to return answers to consuming nodes as soon as the answers become available. As its name implies, Single Stack Scheduling uses a stack to schedule the applicable SLG operations.

This strategy has two distinct mechanisms to return answers to consuming nodes: (1) when a consuming node is created whose selected literal is a variant of a subgoal S which has answers, the consuming node will be resolved against the existing answers as if they were unit clauses; and (2) when a new answer is created for a subgoal S , this answer is scheduled (on the top of the stack) to be immediately returned to all consuming nodes for S .

Using the SLG_{sched} framework, the sequence combinator \oplus for Single Stack Scheduling can be defined as follows.

Definition 4.1 ($\oplus_{SingleStack}$) Let $\sigma = op_1, op_2, \dots, op_n, n \geq 0$, be a scheduling sequence for forest \mathcal{F} , and op_{new} any SLG_{sched} operation (SC, PCR, ACR, Compl). Then,

- If $op_{new} = Compl(NodeNumber)$, and $op_k = Compl(LeaderNode(NodeNumber, \mathcal{F}))$,

$$\oplus_{Batched}(op_{new}, \sigma) = op_1, \dots, op_{k-1}, \underline{op_{new}}, op_k, \dots, op_{i_n}$$

- Otherwise,

$$op_{new} \oplus_{SingleStack} \sigma = \underline{op_{new}}, \sigma.$$

□

Example 4.1 Given the following left-recursive transitive closure and graph

```
:- table p/2.
(C0) a(1,2).
(C1) a(2,3).
(C2) a(1,4).
(C3) p(X,Y) :- p(X,Z), a(Z,Y).
(C4) p(X,Y) :- a(X,Y).
```

⁴Due to space limitations, we restrict ourselves to definite programs.

$\sigma_0 = \underline{SC(0)}$
 $\sigma_1 = \underline{PCR(1,C3), PCR(1,C4), Compl^i(1)}$
 $\sigma_2 = \underline{PCR(1,C4), Compl^i(1)}$
 $\sigma_3 = \underline{PCR(3,C0), PCR(3,C1), PCR(3,C2), Compl^i(1)}$
 $\sigma_4 = \underline{ACR(4,0), ACR(4,2), PCR(3,C1), PCR(3,C2), Compl^i(1)}$
 $\sigma_5 = \underline{ACR(4,2), PCR(3,C1), PCR(3,C2), Compl^i(1)}$
 $\sigma_6 = \underline{PCR(6,C1), PCR(3,C1), PCR(3,C2), Compl^i(1)}$
 $\sigma_7 = \underline{ACR(7,0), ACR(7,2), PCR(3,C1), PCR(3,C2), Compl^i(1)}$
 $\sigma_8 = \underline{ACR(7,2), PCR(3,C1), PCR(3,C2), Compl^i(1)}$
 $\sigma_9 = \underline{PCR(3,C1), PCR(3,C2), Compl^i(1)}$
 $\sigma_{10} = \underline{ACR(10,0), ACR(10,2), PCR(3,C2), Compl^i(1)}$
 $\sigma_{11} = \underline{ACR(10,2), PCR(3,C2), Compl^i(1)}$
 $\sigma_{12} = \underline{PCR(3,C2), Compl^i(1)}$
 $\sigma_{13} = \underline{ACR(13,0), ACR(13,2), Compl^i(1)}$
 $\sigma_{14} = \underline{ACR(13,2), Compl^i(1)}$
 $\sigma_{15} = \underline{Compl^i(1)}$
 $\sigma_{16} = \emptyset$

Figure 2: Scheduling sequences for Single Stack Scheduling

To illustrate the actions of Single Stack Scheduling, Figure 2 shows the scheduling sequences generated during the evaluation of the query $p(X, Y)$. The operations added to a sequence at each step are underlined. The final SLG forest for this query and program is shown in Figure 3.

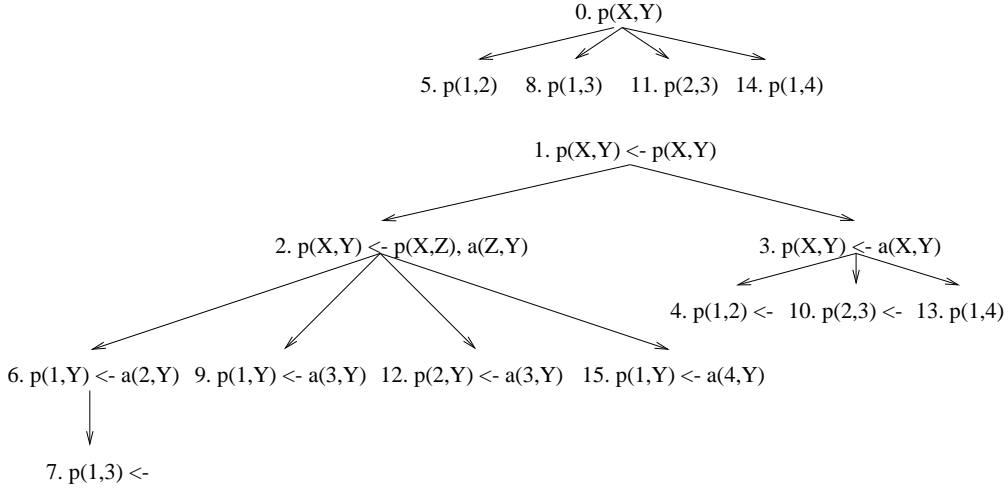


Figure 3: SLG forest for Single Stack Scheduling

Let us examine this example in detail. When the original query is issued, since $p/2$ is a tabled predicate, SUBGOAL CALL is applicable, and $SC(0)$ (where 0 is the number of the consuming node for the top-level query) is *pushed* onto the scheduling sequence σ_0 . When this operation is selected, since the subgoal $p(X, Y)$ is new to the evaluation, a new tree rooted at node 1. $p(X, Y) \leftarrow p(X, Y)$ is created in Figure 3. A completion operation as well as all the available program clause resolution for $p(X, Y)$ are pushed onto the sequence σ_1 (in Figure 2). The next operation to be selected from the scheduling sequence is $PCR(1,C3)$. PROGRAM CLAUSE RESOLUTION creates node 2 in the tree, the result of the resolution of $p(X, Y)$ and its first clause. The selected literal in this new node is tabled, but since there is already a tree for $p(X, Y)$ with no answers, no new operations are applicable. The next operation to be selected is $PCR(1,C4)$. This leads to the creation of node 3, whose selected literal is non-tabled. For non-tabled subgoals, Single Stack Scheduling immediately schedules all available program clause resolution (so as to keep Prolog's search

order), and accordingly, PCR tuples are added to sequence σ_3 . The first clause for the predicate $a/2$ is resolved against node 3, which leads to an answer in node 4. Single Stack Scheduling schedules the return of every new answer to the consuming nodes as soon as the answer is created, thus, $ACR(4,2)$ and $ACR(4,0)$ are pushed onto the sequence. When the answer in node 4 is returned to the consuming node 2, node 6 is created, and as a result, PROGRAM CLAUSE RESOLUTION is scheduled for node 6 ($PCR(6,C1)$ is added to σ_6). When clause C1 is resolved against node 6, a new answer is created (in node 7), which is immediately scheduled to be returned ($ACR(7,0)$, $ACR(7,2)$ are added to σ_7). When the answer in node 7 is returned to node 2, it creates node 9 for which no PROGRAM CLAUSE RESOLUTION is available. The next selected operation in σ_9 , $PCR(3,C1)$, is applied: a new answer is created in node 10 and ANSWER CLAUSE RESOLUTION of this answer against the consuming nodes is scheduled. The last clause for $a/2$ is tried in σ_{12} , and finally the only operation left in the sequence is $Compl(1)$. \square

We provide a brief discussion of the fairness of $\oplus_{SingleStack}$. For programs with bounded-term depth, only a finite number of different scheduling tuples can be added to the scheduling sequence (otherwise, the corresponding SLG evaluation would not terminate). Therefore we need only ensure that $\oplus_{SingleStack}$ does not repeatedly add the same scheduling tuple to the scheduling sequence. But this can not be the case for $\oplus_{SingleStack}$. For definite programs, the only operation that is pushed back into the scheduling sequence in case it is not applicable at the time it is selected is $Compl(NodeNumber)$ (when the tree rooted at $NodeNumber: S \leftarrow S$ is not completely evaluated). In this case, case $\oplus_{SingleStack}$ adds this tuple right before the $Compl$ tuple for L , the leader of the SCC S lies in. Since all other operations are always added and consumed from the head of the sequence, when the $Compl$ tuple for L is taken from the head of the sequence, the SCC will be completely evaluated. Thus the next time $Compl(NodeNumber)$ is selected, it is guaranteed that S is completely evaluated and thus it will be removed from the sequence (see Definition A.1).

In what follows we will give the definitions for the sequence combinators of other scheduling strategies and briefly describe their main characteristics. Detailed examples and proof of fairness for these sequence combinators can be found in the full version of this paper.

4.2 Batched Scheduling

Even though Single Stack Scheduling is conceptually simple, a WAM-based implementation of Single Stack Scheduling has a number of drawbacks, most notably in memory usage (for more details see [?]). Batched Scheduling addresses these drawbacks by taking advantage of engine-level optimizations to reduce stack freezes, and trailing and untrailing. Because of this motivation, it might at first sight appear difficult to derive a sequence combinator to describe Batched Scheduling.

Batched Scheduling discriminates between two types of consuming nodes when returning answers: the consuming nodes in the calling environments (this property can be obtained for a given indexed forest, see Section 2.1) of the original call to a subgoal and consuming nodes that resulted from repeated calls to a subgoal. The \oplus sequence combinator for Batched Scheduling is defined below. For clarity of presentation we make use of two functions: *GenNode*, that given a forest \mathcal{F} and consuming node N with selected literal S (or $\text{not}(S)$) in \mathcal{F} , returns the index of the root node of the tree for S ; and *LeaderNode*, that given a root node $Root$ and a forest, returns the index of the root subgoal that is the leader of $Root$'s SCC. We also denote the scheduling tuple $ACR(Node, Answer)$ as ACR_{ce} if $Node$ is the calling environment for its selected literal, and as ACR_{cns} otherwise.

Definition 4.2 ($\oplus_{Batched}$) Let $\sigma = op_1, op_2, \dots, op_n$, $n \geq 0$, be the scheduling sequence for the forest \mathcal{F} , and op_{new} a new applicable operation. The function $\oplus_{Batched}$ can be defined as follows:

- If $\sigma = \emptyset$, $\oplus_{Batched}(op_{new}, \sigma) = \underline{op_{new}}$
- If $op_{new} \in \{SC, PCR, Compl^i, ACR_{ce}\}$, $\oplus_{Batched}(op_{new}, \sigma) = \underline{op_{new}}, \sigma$
- If $op_{new} = ACR_{cns}(NodeNumber)$, and $op_k = Compl(GenNode(NodeNumber, \mathcal{F}))$,

$$\oplus_{Batched}(op_{new}, \sigma) = op_1, \dots, op_{k-1}, \underline{op_{new}}, op_k, \dots, op_n$$

- If $op_{new} = Compl^*(NodeNumber)$, and $op_k = Compl(LeaderNode(NodeNumber, \mathcal{F}))$,

$$\oplus_{Batched}(op_{new}, \sigma) = op_1, \dots, op_{k-1}, \underline{op_{new}}, op_k, \dots, op_{i_n}$$

□

Contrasting Definitions 4.1 and 4.2, we can see that the main difference between Single Stack Scheduling and Batched Scheduling lies in the scheduling of ACR_{cns} tuples. Instead of adding ACR_{cns} to the head of the sequence (so that the answer can be immediately returned), $\oplus_{batched}$ adds the ACR_{cns} for a subgoal S right before the $Compl$ tuple for S — in fact delaying the return of the answer to the consuming node. In practice, delaying the return of answers to consuming nodes reduces the need to freeze and the need to move around the forest to return answers to consuming nodes. This scheduling change results, at the implementation level, in a more efficient evaluation engine than Single Stack Scheduling.

4.3 Local Scheduling

As presented in [?], Local Scheduling can be thought of as a variation of Batched Scheduling that evaluates one strongly connected component at a time. It achieves this order of evaluation by ensuring answers for a subgoal S are only returned to its calling environment either after S is completely evaluated, or after S is found not to be the leader of its SCC. In [?] we have shown that because Local Scheduling is able to avoid non-productive computation in the presence of answer subsumption, it can perform exponentially better than Batched Scheduling for problems such as aggregate computation and program analysis. Local Scheduling can also benefit the evaluation of programs with negation. Local Scheduling can be described by the \oplus_{Local} combinator defined below.

Definition 4.3 (\oplus_{Local}) Given a scheduling sequence $\sigma = op_1, op_2, \dots, op_n$ for the forest \mathcal{F} , where $n \geq 0$, and a new operation op_{new} . The function \oplus_{Local} can be defined as follows:

- If $\sigma = \emptyset$, $\oplus_{Local}(op_{new}, \sigma) = \underline{op_{new}}$
- If $op_{new} \in \{SC, PCR, Compl^i\}$, $\oplus_{Local}(op_{new}, \sigma) = \underline{op_{new}}, \sigma$
- If $op_{new} = Compl^*(NodeNumber)$, and $op_k = Compl(LeaderNode(NodeNumber, \mathcal{F}))$,

$$\oplus_{Local}(op_{new}, \sigma) = op_1, \dots, op_{k-1}, \underline{op_{new}}, op_k, \dots, op_n$$

- If $op_{new} = ACR_{cns}(NodeNumber)$, and $op_k = Compl(GenNode(NodeNumber, \mathcal{F}))$,

$$\oplus_{Local}(op_{new}, \sigma) = op_1, \dots, op_{k-1}, \underline{op_{new}}, op_k, \dots, op_{i_n}$$

- If $op_{new} = \{ACR_{ce}(NodeNumber)\}$, and $op_k = Compl(GenNode(NodeNumber, \mathcal{F}))$,

$$\oplus_{Local}(op_{new}, \sigma) = op_1, \dots, op_k, \underline{op_{new}}, op_{k+1}, \dots, op_n$$

□

We can see, from the definition of \oplus_{Local} , that the SCC ordering in the evaluation is enforced by scheduling answers to be returned to the calling environment (ACR_{ce}) of a subgoal S after the $Compl$ tuple for S . Therefore, if S is the leader of an SCC, answers will only be returned to its calling environment after it is completely evaluated. Combining this property with answer subsumption can arbitrarily improve the performance of some programs. Answer subsumption can be performed when a new answer is created. While adding an answer, the evaluation may check whether the new answer is more general than those currently in the forest. If it is more general, this new answer is added and the subsumed answers are removed. Because Local Scheduling delays the return of answers out of an SCC until the SCC is completely evaluated, only the most general answers are returned out of the SCC and non-productive computation (using non-optimal answers) is thus avoided.

4.4 Breadth-First Scheduling

As described in [?], Breadth-First Scheduling enforces *fairness* across fixpoint iterations of an evaluation, that is, subgoals called in an iteration i can only be executed in iteration $i + 1$, and answers created in iteration i can only be returned in iteration $i + 1$. Iterations are controlled by the breadth-first leader, the *bfleader*, the first tabled subgoal called in the evaluation — the end of an iteration is marked when $\text{Compl}(\text{bfleader})$ is selected from the scheduling sequence.

In general, new operations are added to the end of the scheduling sequence — which, for Breadth-First Scheduling can be seen (mostly) as a queue. There are a few exceptions to this rule. PROGRAM CLAUSE RESOLUTION, for example, should be added to the beginning of the sequence when it becomes applicable. The reason being that PROGRAM CLAUSE RESOLUTION is scheduled in two situations: (1) at SUBGOAL CALL for subgoals that are new to the evaluation, and (2) when the selected literal of a node is not tabled. In Breadth-First Scheduling when a tabled predicate that is new to the evaluation is selected, $\text{SC}(S)$ is added to the tail of the scheduling sequence, guaranteeing that this subgoal is only executed in the following iteration. However, when $\text{SC}(S)$ is selected, to ensure that S is indeed expanded in the current iteration (as it is supposed to), any available PROGRAM CLAUSE RESOLUTION should be scheduled for immediate execution. To understand the second case, note that non-tabled predicates are evaluated under SLD, and as such, require a depth-first (non-fair) strategy. Thus, any PROGRAM CLAUSE RESOLUTION for non-tabled predicates should be added to the head of the scheduling sequence.

Besides PROGRAM CLAUSE RESOLUTION, in some situations ANSWER CLAUSE RESOLUTION should also be added to the head of the sequence. If a consuming node for a subgoal S is created in iteration i , and S has answers which were created in previous iterations, denoted by the set Answers_{i-1}^S , ANSWER CLAUSE RESOLUTION should be made immediately available for this new node against answers in Answers_{i-1} . However, answers created in the current iteration, denoted by the set $\delta\text{Answers}^S$, should only be returned to the new consuming node in iteration $i + 1$. In Breadth-First Scheduling this behavior is achieved by making sure that answers in Answers_{i-1}^S in previous iterations are added to the head of the sequence, whereas answers in $\delta\text{Answers}^S$ are added to the tail of the sequence.

Note that at any iteration i , the tuple $\text{Compl}(\text{bfleader})$ separates the operations that should be executed in the current iteration (all operations to the left) from the operations that should be executed in iteration $i + 1$ (all operations to the right). Since for each new answer that is created for a subgoal S , a tuple ACR_{ce} is added to return that answer to the calling environment of S , at any point in an iteration i , answers created in i can be identified by the presence of ACR_{ce} tuples for those answers to the right of $\text{Compl}(\text{bfleader})$ (i.e., $\delta\text{Answers}$ is the set of all answers for which there are ACR_{ce} frames to the right of $\text{Compl}(\text{bfleader})$).

Definition 4.4 ($\oplus_{\text{BreadthFirst}}$) Given a scheduling sequence $\sigma = op_1, op_2, \dots, op_n$ for a forest \mathcal{F} , where $op_n = \text{Compl}(\text{bfleader})$. If a new operation op_{new} is to be added to σ :

- If $\sigma = \emptyset$, $\oplus_{\text{BreadthFirst}}(op_{new}, \sigma) = \underline{op_{new}}$
- If $op_{new} = \text{PCR}(\text{NodeNumber}, \text{Clause})$, $\oplus_{\text{BreadthFirst}}(op_{new}, \sigma) = \underline{op_{new}}, \sigma$
- If $op_{new} = \text{ACR}_{cns}(\text{AnswerNumber}, \text{CnsNode})$, and $\text{AnswerNumber} \in \delta\text{Answers}$,

$$\oplus_{\text{BreadthFirst}}(op_{new}, \sigma) = \sigma, \underline{op_{new}}$$

otherwise, if $\text{AnswerNumber} \notin \delta\text{Answers}$,

$$\oplus_{\text{BreadthFirst}}(op_{new}, \sigma) = \underline{op_{new}}, \sigma$$

- If $op_{new} \in \{\text{SC}, \text{ACR}_{ce}, \text{Compl}\}$,

$$\oplus_{\text{BreadthFirst}}(op_{new}, \sigma) = op_1, \dots, op_{n-1}, \text{Compl}(\text{bfleader}), \underline{op_{new}}$$

□

In [?] we presented an implementation of Breadth-First Scheduling, which was proved to be *iteration equivalent* to the semi-naive evaluation of a Magic-transformed (SNMT) [?] program, that is, each iteration Breadth-First Scheduling produces the same information as SNMT. Here we re-state that result using the SLG_{sched} definition of Breadth-First Scheduling.

Theorem 4.1 Let P be a definite program and let $M(P,Q)$ be its magic rewrite for a query Q and $T(P)$ be the fully tabled program, and assume $SC(Q)$ is an element of σ_0 . Then, at each iteration t :

1. An SNMT evaluation of P for Q derives a non-magic fact A iff Breadth-First Scheduling adds a tuple $ACR(A,Cns)$ to the scheduling sequence for some consumer node Cns .
2. An SNMT evaluation of P for Q derives a fact $magic(S)$ iff Breadth-First Scheduling adds a tuple $SC(S)$ to the scheduling sequence.

Proof: The proof is given in the Appendix.

◇

5 Conclusions and Future Work

Scheduling has been recently recognized as a crucial component of tabled evaluations, since the efficiency of an evaluation is highly sensitive to the order in which operations are performed. The empiric results of [?], [?] and [?] highlighted some of the characteristics of specific scheduling strategies. However, the lack of a formal structure made it difficult to define these strategies more precisely, prove their correctness and compare different strategies. Here, we address this problem by proposing a formal framework to model scheduling strategies, SLG_{sched} .

We have used SLG_{sched} to describe, for definite programs, a number of strategies that have been implemented in the XSB system. As for future work, we would like to extend these definitions to include normal programs. We would also like to explore using SLG_{sched} as a means to compare the asymptotic characteristics of various different strategies.

A SLG_{sched} Operations

In some of the following scheduling operations we will make use of three algorithms: *ScheduleNewNode()*, described in Procedure A.1, which given a new node, adds to the scheduling sequence any operations that become applicable as a result of the creation of this new node; Procedure A.2 which adds to the scheduling sequence any operations that become applicable when a new consuming node is created; and *Complete()*, described in A.3, which given a root node, sets the status of this node to completed, and schedules any simplification that might be triggered by the completion of this root node.

Definition A.1 (SLG operations augmented with scheduling) Let \mathcal{F}_n be an indexed forest; $Seq_n = op_1, \dots, op_k$; $Op = op_1$ a scheduling tuple; and \oplus a sequece combinator, which produces a scheduling sequence Seq_{n+1} defined as follows. Note that below we denote by Seq'_n the sequence that results from removing op_1 from Seq_n , that is, $Seq'_n = Seq_n - op_1$.

1. $Op = SC(NodeNumber)$: Given the non-root node $NodeNumber$: $AnswerTemplate \leftarrow DelaySet \mid G, GoalList$, where G is the selected literal S or $not(S)$.

- If S is not contained in \mathcal{F}_n , a new tree with root $n+1$: $S \leftarrow \mid S$ is created; and if $G = S$

$$Seq_{n+1} = Seq'_n \oplus Compl^i(N) \oplus PCR(n+1, Clause_i),$$

for each i such that the head of $Clause_i$ unifies with S .

otherwise, if $G = not(S)$

$$Seq_{n+1} = Seq'_n \oplus Compl^i(N) \oplus ND(N) \oplus$$

$$PCR(n+1, Clause_i), \text{ for each } i \text{ such that the head of } Clause_i \text{ unifies with } S.$$

- If $S \leftarrow S$ is the root of a tree in \mathcal{F}_n :

$$Seq_{n+1} = Seq'_n \oplus ScheduleCnsNode(n+1)$$

2. $Op = PCR(NodeNumber, ClauseNumber)$: If a PROGRAM CLAUSE RESOLUTION operation is still applicable in \mathcal{F} for $NodeNumber$ and $ClauseNumber$, let $n+1$ designate the index of the child of $NodeNumber$ produced by this operation.

$$Seq_{n+1} = Seq'_n \oplus ScheduleNewNode(n+1)$$

3. $Op = ACR(AnswerNodeNumber, NodeNumber)$: Let $AnswerLeafNumber$ be the leaf node of the branch rooted at $AnswerNodeNumber$ (possibly, $AnswerLeafNumber = AnswerNodeNumber$). If an ANSWER CLAUSE RESOLUTION operation is still applicable to $AnswerLeafNumber$ and $NodeNumber$, let $n+1$ designate the index of the child of $NodeNumber$ produced by this operation. Then

$$Seq_{n+1} = Seq'_n \oplus ScheduleNewNode(n+1)$$

4. $Op = ND(NodeNumber)$: Suppose $NodeNumber$ indexes a non-root node $AnswerTemplate \leftarrow DelaySet \mid not(S), G_1, \dots, G_k$.

- If the tree rooted at S is not completed and has no unconditional answers, add $n+1$: $AnswerTemplate \leftarrow DelaySet, S \mid G_1, \dots, G_k$ and

$$Seq_{n+1} = Seq'_n \oplus ScheduleNewNode(n+1)$$

- Otherwise, if S has an unconditional answer

$$\boxed{Seq_{n+1} = Seq'_n \oplus NR(NodeNumber, Fail)}$$

Or if S is completed and has no answers

$$\boxed{Seq_{n+1} = Seq'_n \oplus NR(NodeNumber, Succeed)}$$

5. $Op = NR(NodeNumber, Type)$: Given the non-root node $NodeNumber: AnswerTemplate \leftarrow DelaySet \mid not(S), G_1, \dots, G_k$.

- (Negation Failure) If $Type = Fail$, add a failed child to $NodeNumber$.
- (Negation Success) If $Type = Succeed$, and $NodeNumber$ has no child of the form $n+1: AnswerTemplate \leftarrow DelaySet \mid G_1, \dots, G_k$, add this node as a child of $NodeNumber$ and

$$\boxed{Seq_{n+1} = Seq'_n \oplus ScheduleNewNode(n+1)}$$

6. $Op = Compl(NodeNumber)$: Given a root node $NodeNumber: S \leftarrow \mid S$, and a set $Subg$ of completely evaluated (see Definition 2.4) subgoals such that $S \in Subg$, then set the status of S to *complete* and

$$\boxed{\begin{array}{l} ComplSeq = Complete(S) \\ \text{If there exists a set of unsupported answers for the subgoals in } Subg, \\ Seq_{n+1} = Seq'_n \oplus AnsCompl(Subg) \end{array}}$$

Otherwise, if S is not completely evaluated:

$$\boxed{Seq_{n+1} = Seq'_n \oplus Compl^*(NodeNumber)}$$

7. $Op = Simpl(RootNumber, CondAnswer, Type)$: Let $CondLeaf$ be the leaf node of the branch rooted at $CondAnswer$. If $CondLeaf$ is a failed node, no action needs to be taken, and $Seq_{n+1} = Seq_n$. Otherwise, let $CondLeaf$ be an answer $CondLeaf: AnswerTemplate \leftarrow DelaySet \mid$ in a tree whose root has the form $S' \leftarrow S'$, and let $RootNumber$ index a root node: $S \leftarrow S$:

(a) (Negative simplification) $Type = Negative$: In this case, $not(S) \in DelaySet$.

- i. If S is failed, create a child of $CondLeaf$ with $not(S)$ removed from $DelaySet$, and if this new child node has an empty delay set:

$$\boxed{Seq_{n+1} = Seq'_n \oplus Simpl(RootNumber, Answer_p, Positive) \oplus Simpl(RootNumber, Answer_n, Negative)}$$

for each $Answer_p$ that contains a delay literal $L_{AnswerTemplate}^{S'}$ in its delay set, and for each $Answer_n$ that contains $not(AnswerTemplate)$ in its delay set.

- ii. Otherwise, if there are unconditional answers in the tree of S , create a failed child for node $CondLeaf$ — and if $CondLeaf$ was the last answer for its root subgoal $Root_{CondLeaf}$,

$$\boxed{Seq_{n+1} = Seq'_n \oplus AnsCompl(\{Root_{CondLeaf}\}) \oplus Simpl(RootNumber, Answer_p, Positive) \oplus Simpl(RootNumber, Answer_n, Negative)}$$

for each $Answer_p$ that contains a delayed literal $L_{AnswerTemplate}^{S'}$ in its delay set, and for each $Answer_n$ that contains $not(AnswerTemplate)$ in its delay set.

- (b) (Positive simplification) $Type = Positive$: In this case $D_{SAns}^S \in DelaySet$. If there is an unconditional answer A in the tree rooted at S , create a new child node for $CondLeaf$ with D_{SAns}^S removed from the delay list. If this new child node has an empty delay set:

$$\boxed{Seq_{n+1} = Seq'_n \oplus Simpl(RootNumber, Answer_p, Positive) \oplus Simpl(RootNumber, Answer_n, Negative)}$$

Procedure A.2 ScheduleCnsNode(N)

```
1  TmpSeq =  $\emptyset$ ;  
2  Given  $N$ : AnswerTemplate  $\leftarrow$  DelaySet  $|L, G_1, \dots, G_n$   
3  If  $L = S$ , for all  $i$  such that Answer $_i$  is an answer in the tree rooted at  $S \leftarrow S$   
4     $\boxed{\text{TmpSeq} = \text{TmpSeq} \oplus \text{ACR}(N, \text{Answer}_i)}$   
5  Else if  $L = \text{not}(S)$   
6    If the tree with root  $S \leftarrow S$  has no unconditional answers  
7     $\boxed{\text{TmpSeq} = \text{ND}(N)}$   
8    If the tree with root  $S \leftarrow S$  has unconditional answers  
9     $\boxed{\text{TmpSeq} = \text{NR}(N, \text{Fail})}$   
10   If the tree with root  $S \leftarrow S$  has status completed and no answers  
11    $\boxed{\text{TmpSeq} = \text{NR}(N, \text{Succeed})}$   
12  Return TmpSeq;
```

Procedure A.3 Complete(RootNumber)

```
1  TmpSeq =  $\emptyset$ ;  
2  For all answer nodes CondAnswer  $\leftarrow$  DelaySet in  $\mathcal{F}_n$   
3    If  $D_{S_{Ans}}^S$  is in the delay list of CondAnswer  
4     $\boxed{\text{TmpSeq} = \text{TmpSeq} \oplus \text{Simpl}(\text{RootNumber}, \text{CondAnswer}, \text{Positive})}$   
5    else if  $S$  is in the delay list of CondAnswer  
6     $\boxed{\text{TmpSeq} = \text{TmpSeq} \oplus \text{Simpl}(\text{RootNumber}, \text{CondAnswer}, \text{Negative})}$   
7  Return TmpSeq;
```

B Auxiliary Material for Referees

B.1 Definitions of SLG

Definition B.1 (SLG Evaluation) Given a finite non-floundering program P , an SLG evaluation \mathcal{E} for a tabled subgoal $root$ is a sequence of forests $\mathcal{F}_0, \mathcal{F}_1, \dots, \mathcal{F}_n$, such that:

- \mathcal{F}_0 is the forest containing the initial query;
- For each finite ordinal i ($i \geq 0$), \mathcal{F}_{i+1} is obtained from \mathcal{F}_i by an application of one of the SLG operations (see Definition B.2).

If no operation is applicable (see Definition B.2) to \mathcal{F}_n , \mathcal{F}_n is called a *final forest* of \mathcal{E} . □

The following definitions of SLG operations are a variant of those presented in [?] and are used in the proofs of Theorem 3.1.

Definition B.2 (SLG Operations) Given a forest \mathcal{F}_n of an SLG evaluation of program P and query Q , \mathcal{F}_{n+1} may be produced by one of the following operations.

1. **SUBGOAL CALL:** Let \mathcal{F}_n contain a non-root node $N = Ans \leftarrow DelaySet | GoalList$ where G is the selected literal S or $not(S)$. Assume \mathcal{F}_n does not contain S . Then we say that a **NEW SUBGOAL** operation is applicable for S , and its action is to add the tree $S \leftarrow | S$ to \mathcal{F}_n .
2. **PROGRAM CLAUSE RESOLUTION:** Let \mathcal{F}_n contain a root node $N = S \leftarrow | S$ or a non-root node $AnswerTemplate \leftarrow Delayset | Goallist$, where S is the selected literal of $Goallist$. Let C be a program clause $Head \leftarrow Body$ such that $Head$ unifies with S with mgu θ . Assume that in \mathcal{F}_n , N does not have a child $N_{child} = (S \leftarrow | Body)\theta$. Then we say that a **PROGRAM CLAUSE RESOLUTION** operation is applicable for N and C and its action is to add N_{child} as a child of N .
3. **ANSWER CLAUSE RESOLUTION:** Let \mathcal{F}_n contain a non-root node N whose selected literal S is positive. Let Ans be an answer node for S in \mathcal{F}_n and N' be the SLG resolvent of N and Ans on S . Assume that in \mathcal{F}_n , N does not have a child N' . Then we say that an **ANSWER CLAUSE RESOLUTION** operation is applicable to N and Ans and its action is to add N' as a child of N .
4. **NEGATIVE DELAY:** Let \mathcal{F}_n contain a leaf node $N = Ans \leftarrow DelaySet | GoalList$, whose selected literal is $not S$. Assume that S is contained in \mathcal{F} but has not been completely evaluated. Then we say that a **NEGATIVE DELAY** operation is applicable to N and its action is to create a child for N of the form $Ans \leftarrow DelaySet, not(S)^S | GoalList$.
5. **NEGATIVE RETURN:** Let \mathcal{F}_n contain a leaf node $N = Ans \leftarrow DelaySet | GoalList$ with selected literal $not S$.
 - (a) **NEGATION SUCCESS:** If S is failed in \mathcal{F} , then we say that a **NEGATIVE RETURN** operation is applicable to N and its action is to create a child for N of the form $Ans \leftarrow DelaySet | GoalList$.
 - (b) **NEGATION FAILURE:** If S succeeds in \mathcal{F} , then we say that a **NEGATIVE RETURN** operation is applicable to N and its action is to create a child for N of the form $fail$.
6. **SIMPLIFICATION:** Let \mathcal{F}_n contain a leaf node $N = Ans \leftarrow DelaySet |$, and let $L \in Delayset$
 - (a) If L is failed in \mathcal{F} then we say that a **simplification** operation is applicable for N and L and its action is to create a child $fail$ for N .
 - (b) If L is successful in \mathcal{F} , then we say that a **simplification** operation is applicable for N and L and its action is to create a child $Ans \leftarrow DelaySet'$ for N , where $DelaySet' = DelaySet - L$.

7. COMPLETION: Given a completely evaluated set \mathcal{S} of subgoals (Definition 2.4) we say that a COMPLETION operation is applicable for \mathcal{S} , and its action is to mark the trees for all subgoals in \mathcal{S} as completed.
8. ANSWER COMPLETION: Given a set of unsupported answers \mathcal{UA} , then we say that an ANSWER COMPLETION operation is applicable to \mathcal{UA} and its action is to create a failed child for each answer $Ans \in \mathcal{UA}$.

□

B.2 SLG_{sched} : Proofs of Soundness and Completeness

Before proving soundness and completeness of SLG_{sched} with respect to SLG, we provide supporting definitions. Applicable SLG operations are represented in a scheduling sequence in the following manner.

Definition B.3 (Scheduling Tuples) Scheduling tuples have the form:

- SC(NodeNumber): apply SUBGOAL CALL to the selected literal of node NodeNumber;
- PCR(NodeNumber,ClauseNumber): resolve the clause with ClauseNumber against the selected literal of node NodeNumber;
- ACR(NodeNumber,AnswerNumber): resolve the answer node AnswerNumber against the selected literal of node NodeNumber. We distinguish between the acts of returning an answer to the calling environment of a subgoal (denoted by ACR_{ce}) and that of returning an answer to a consuming node of a subgoal (denoted by ACR_{cns});⁵
- ND(NodeNumber): apply NEGATIVE DELAY to the selected literal of NodeNumber;
- NR(NodeNumber,Type): apply NEGATIVE RETURN to the selected literal of NodeNumber, and type indicates whether that literal succeeds or fails;
- Compl(NodeNumber): apply COMPLETION to the node NodeNumber. As we will see later, a Compl frame for a subgoal can be added to the scheduling sequence multiple times. We distinguish the first occurrence of the completion frame for a subgoal by tagging it with an i (for initial), $Compl^i$; the later occurrences are tagged with an asterisk, $Compl^*$;
- Simpl(NodeNumber,CondAnswer,Type): given the root node with number NodeNumber and status completed, and an answer CondAnswer, simplify the atoms in the delay list of CondAnswer that are variants of node NodeNumber. Type indicates whether positive or negative simplification will be applied;
- AnsCompl(NodeSet): if there is a set of unsupported answers for a set of completely evaluated subgoals $Subg$, such that $NodeSet \subseteq Subg$, apply ANSWER COMPLETION to some unsupported answers in trees rooted at nodes in $Subg$.

□

Theorem 3.1 follows immediately from the following lemmas.

Lemma B.1 *Let \mathcal{E}' be a non-floundering terminating SLG_{sched} evaluation of a given program P and query Q , such that*

$$\mathcal{E}' = \{(\mathcal{F}_{i_0}, Seq_0), (\mathcal{F}_{i_1}, Seq_1), \dots, (\mathcal{F}_{i_m}, Seq_m)\}$$

then there exists an SLG evaluation $\mathcal{E} = \{\mathcal{F}_0, \mathcal{F}_1, \dots, \mathcal{F}_n\}$ for P and Q , such that $\forall k \in \{0, \dots, n\}, \exists i_{k'} \in \{0, \dots, m\}, i_{k'} \geq k$ such that $\mathcal{F}_k = \mathcal{F}_{i_{k'}}$ (i.e., for an SLG evaluation $\mathcal{E} = \{\mathcal{F}_0, \mathcal{F}_1, \dots, \mathcal{F}_n\}$, the corresponding SLG_{sched} evaluation might contain multiple tuples with the same forest, for instance, $\mathcal{E}' = \{(\mathcal{F}_0, Seq_0), \dots, (\mathcal{F}_j, Seq_k), (\mathcal{F}_j, Seq_{k+1}), \dots, (\mathcal{F}_{i_m}, Seq_m)\}$).

⁵Throughout the text, we use ACR to denote either ACR_{ce} or ACR_{cns} , and similarly, Compl to denote either $Compl^i$ or $Compl^*$.

Proof: (sketch)

Proof is by a simple induction based on the following observations. Note that in SLG_{sched} , an SLG forest $\mathcal{F}_{i_{k'+1}}$ is produced from a forest $\mathcal{F}_{i_{k'}}$ by applying an SLG_{sched} operation op of Definition A.1, at which point new applicable operations might be added to the sequence. In general there are two cases: op may be applicable to $\mathcal{F}_{i_{k'}}$ or it will not be. It can be seen from examination of Definitions A.1 and B.2 that if an SLG_{sched} operation is applicable to an indexed forest \mathcal{F}_n , the corresponding SLG operation will be applicable to the equivalent non-indexed forest. On the other hand, if the SLG_{sched} operation op is not applicable to \mathcal{F}_n , the corresponding SLG operation will not be applicable either. In these cases, by Definition A.1, \mathcal{F}_{n+1} will equal \mathcal{F}_n . \diamond

Lemma B.2 *Let \mathcal{E} be a finite and non-floundering SLG evaluation of a given program P and query Q , such that $\mathcal{E} = \{\mathcal{F}_0, \mathcal{F}_1, \dots, \mathcal{F}_n\}$. Then there exists a fair sequence combinator \oplus and SLG_{sched} evaluation based on \oplus*

$$\mathcal{E}' = \{(\mathcal{F}_{i_0}, Seq_0), (\mathcal{F}_{i_1}, Seq_1), \dots, (\mathcal{F}_{i_m}, Seq_m)\}$$

such that $\forall k \in \{0, \dots, n\}, \exists k' \in \{0, \dots, m\}, i_{k'} \geq k$ such that $\mathcal{F}_k = \mathcal{F}_{i_{k'}}$, and $Seq_{k'} \supseteq App_{\mathcal{F}_k}$.

Proof: (sketch) We prove by induction that at each point in the evaluation, if an operation is applicable in a forest, it is also present in the scheduling sequence associated with the forest. That is, given a forest $\mathcal{F}_{i_{k'}}$ in an SLG_{sched} evaluation

$$\mathcal{F}_{i_{k'}} \xrightarrow{op} \mathcal{F}_{i_{k'+1}} \Rightarrow op \in Seq_{k'}$$

- *Base case:* Given a program P and query Q , the only applicable SLG operation (Definition B.2) is SUBGOAL CALL, which by definition is in the initial scheduling sequence.
- *Inductive step:* Let us examine each possible applicable operation in the forest \mathcal{F}_n , and prove that if an operation op is applicable in the forest \mathcal{F}_n , then $op \in Seq_n$

($op = \text{SUBGOAL CALL}$) If SUBGOAL CALL is applicable in \mathcal{F}_n , then exists a leaf node $NodeNumber: AT \leftarrow DS \mid S, GL$ (or the original query S) such that S is tabled and there is no tree in \mathcal{F}_n with a root node whose subgoal is a variant of S . We claim that $SC(NodeNumber) \in Seq_n$. By definition, whenever a new node is created in SLG_{sched} , if the selected literal in the new node is tabled and it is not present in the evaluation, $SC(NodeNumber)$ is added to the scheduling sequence (see Procedure A.1).

($op = \text{PROGRAM CLAUSE RESOLUTION}$) If PROGRAM CLAUSE RESOLUTION is applicable, either (1) there is a root node $NodeNumber: S \leftarrow \mid S$; or (2) there is a non-root node $NodeNumber: AT \leftarrow DS \mid S, GL$ in which S is not tabled. In either case there is a program clause whose head unifies with S .

If PROGRAM CLAUSE RESOLUTION is applicable to the root node $NodeNumber$ and a clause C , then $PCR(NodeNumber, C) \in Seq_n$, since the root node was created as the result of SUBGOAL CALL, which in SLG_{sched} adds PCR tuples for the new root against all matching clauses (see Definition A.1 (1)).

If PROGRAM CLAUSE RESOLUTION is applicable to the non-root node $NodeNumber$ and a clause C , then $PCR(NodeNumber, C) \in Seq_n$, since when a new node is created, if the selected literal is non-tabled, PCR tuples for the new root against all matching clauses are added (see Procedure A.1).

($op = \text{ANSWER CLAUSE RESOLUTION}$) If ANSWER CLAUSE RESOLUTION is applicable in \mathcal{F}_n , there must exist an answer node AN in the tree for a subgoal S , and a non-root node CN whose selected literal is a variant of S . We claim that $ACR(AN', CN) \in Seq_n$, where $AN' \leq AN$, and if $AN \neq AN'$, AN' is an ancestor of AN that has an empty goal list. Let us consider the following possibilities:

- $AN' < CN$ (i.e., the answer was created before the consuming node). In this case, when SUBGOAL CALL was executed for node CN , SLG_{sched} (see Definition A.1 1) must have add ACR frames for CN and all answers available in the tree for S , in particular $ACR(AN', CN)$.

- $AN' > CN$ (i.e., the answer was created after the consuming node). When the answer AN' was created, Procedure A.1 (ScheduleNewNode) must have added ACR frames for AN' and all non-root nodes whose selected literals are variants of S , in particular $ACR(AN', CN)$.

In either case, if $AN \neq AN'$, SLG_{sched} uses AN' simply as a designator of the answer AN , and ACR is actually applied to AN (see Definition A.1 (3)).

(op = NEGATIVE DELAY) If NEGATIVE DELAY is applicable, then (1) there exists a leaf node $N: AnswerTemplate \leftarrow DelaySet \mid not(S), GoalList$, and (2) there exists a tree with root S whose status is non-complete.

In SLG_{sched} , when the leaf node N is created whose selected literal S is tabled, a frame $SC(N)$ is added to the scheduling sequence. Since when NEGATIVE DELAY becomes applicable, there must be a tree with root S , we are guaranteed that $SC(N)$ has already been selected, and when it was selected it must have added $ND(N)$ to the scheduling sequence — by Definition A.1 (1), if a subgoal is called negatively and it is either new to the evaluation, or if there is a non-completed tree with that subgoal as root, ND is scheduled, and thus $ND(N) \in Seq_n$.

(op = NEGATIVE RETURN) If NEGATIVE RETURN is applicable, there exists a node $N: AnswerTemplate \leftarrow DelaySet \mid not(S), GoalList$, and (1) S is completed and has no answers, or (2) S has an unconditional answer. In either case, the tabled subgoal S must have already been called, that is, $SC(N)$ has been selected. Then, by Definition A.1 (1) $NR(N) \in Seq_n$, since unconditional answers can never become conditional, and completed subgoals “uncompleted”.

(op = COMPLETION) In COMPLETION is applicable to \mathcal{F}_n there must exist a set $Subg$ of subgoals which is completely evaluated. We claim that the tuple $Compl(Subg_i) \in Seq_n$, for each subgoal i in $Subg$. In SLG_{sched} , a Compl operation is scheduled for each new subgoal at SUBGOAL CALL. Even though Compl for a subgoal S is scheduled at subgoal call, it will only be applicable after all operations (SC, PCR, ACR, ND and NR) involving S and/or subgoals S depends on have been applied. In order to ensure $Compl(S)$ is present in the sequence whenever it becomes applicable, SLG_{sched} reschedules this operation in case it is selected early.

(op = SIMPLIFICATION) If SIMPLIFICATION is applicable to a conditional answer $AN: Ans \leftarrow DS$, then there is an atom $S \in DS$, or $A_{A'}^S \in DS$ whose truth value is known.

Notice that the truth value of an atom becomes known when: (1) there is an unconditional answer for the subgoal S that is a variant of A ; (2) S is completed without any unconditional answers; or (3) the last conditional answer of S is failed (see Definition B.2). In any of these cases SLG_{sched} schedules the simplification of AN .

The atom S (or $A_{A'}^S$) is only delayed if S is not completed and has no unconditional answers. If S is later completed, SLG_{sched} schedules simplification for all answers that contain a variant of S in their delay set (see Procedure A.3). Alternatively, S can only get unconditional answers if one of its conditional answers become unconditional, and this only happens as a result of SIMPLIFICATION, in which case SLG_{sched} schedules all applicable simplifications (see Definition A.1 (7)). Finally, the last conditional answer of a subgoal can only become failed as a result of SIMPLIFICATION or ANSWER COMPLETION, and in both cases SLG_{sched} schedules all applicable simplifications (see Definition A.1 (7) and (8)).

(op = ANSWER COMPLETION) If ANSWER COMPLETION is applicable then there exists a set of unsupported answers. We argue that, in SLG_{sched} , if there is a set UA of unsupported answers, then $AnsCompl(UA) \in Seq_n$, where $UA \subseteq \bigcup UA_i$. A set of answers can only become unsupported if a subgoal gets completed, as the result of SIMPLIFICATION, or ANSWER COMPLETION (see Definition B.2). In either of these cases SLG_{sched} schedules any available ANSWER COMPLETION (see Definition A.1 (6), (7) and (8)).

◇

Proof of Theorem 4.1

Some notation will be useful to prove Theorem 4.1:

- $\delta Subgoals_t$ is the set of non-completed tabled subgoals added during an iteration t (it is the set of subgoals for which there are SC tuples to the right of Compl(bfleader));
- δCns_t^S is the set of consuming nodes in tree S added during iteration t (i.e., $AT \leftarrow GoalList$ might be in δCns_t^S for some t);
- δAns_t^S is the set of answers that have been added to tree S during iteration t (or equivalently, the set of all answers for which there are ACR_{ce} tuples to the right of Compl(bfleader)).

The sets of all answers, consuming nodes and subgoals for S at t can be constructed by taking the union of the deltas for S over all scheduling sequences for iterations than or equal to t . We denote these sets as Ans_t^S , Cns_t^S and $Subgoals_t$, respectively.

Definition B.4 (Magic Templates Rewriting [?]) Let P be a program and let Q be a query to P . The *magic rewrite* of P for Q , or $M(P, Q)$, is constructed as follows.

1. Create a seed fact $magic(Q)$.
2. For each rule R in P add the *modified version* of the rule to $M(P, Q)$. If a rule R has head $p(\vec{X})$, the modified version is obtained by adding the literal $magic(p(\vec{X}))$ to the body of R .
3. For each rule R in P with head $p(\vec{X})$, and for each occurrence of a derived literal $q_i(\vec{Y})$ in its body, add a *query rule* to $M(P, Q)$ whose head is $magic(q_i(\vec{Y}))$ and whose body contains the literal $magic(p(\vec{X}))$ and all literals preceding $q_i(\vec{Y})$ in R .

□

Theorem 4.1 Let P be a definite program and let $M(P, Q)$ be its magic rewrite for a query Q and $T(P)$ be the fully tabled program, and assume $SC(Q)$ is an element of σ_0 . Then, at each iteration t :

1. An SNMT evaluation of P for Q derives a non-magic fact A iff Breadth-First Scheduling adds a tuple $ACR_{ce}(A, Cns)$ to the scheduling sequence for some consumer node Cns .
2. An SNMT evaluation of P for Q derives a fact $magic(S)$ iff Breadth-First Scheduling adds a tuple $SC(S)$ to a scheduling sequence σ_i for some i .

Proof:

Statements 1 and 2 are proven together by induction on the number of iterations of the Breadth-First evaluation. In order to simplify the presentation, we will use subgoal names in place of their corresponding node numbers in the SLG_{sched} operations.

- Base case.
 - 1) *Non-magic Facts*: Suppose the first iteration of the Breadth-First Scheduling evaluation produces an answer fact A via rule R , that is, a new answer is created as the result of $PCR(S, R)$, where A is a variant of S . The production of any answer in the Breadth-First Scheduling evaluation depends on the presence of a corresponding subgoal in the evaluation. Since Q is the only subgoal in $Subgoals_0$ and since Ans_0 is empty, A must have root subgoal Q and cannot depend on the results of any other tabled subgoals, that is A can depend only on base facts and not on derived facts. Under these conditions, A would be also produced by the SNMT evaluation.

- 2) *Magic Facts*: Suppose the Breadth-First Scheduling evaluation adds a tuple $SC(S)$. In this case, there is a rule R_1 whose head unifies with Q , and a prefix Lit_1, \dots, Lit_m leading a call to a rule R_2 whose head unifies with S . Also, it is clear that each Lit_i must be a base fact. Next in the Breadth-First Scheduling evaluation, R_2 is used for resolution, and $SC(S)$ is immediately produced. Now consider how R_2 is produced by rewrite rule 3 of Definition B.4:

$$R_1 : Q \leftarrow Lit_1, \dots, Lit_m, S \quad R_2 : S \leftarrow Body_S \quad \Rightarrow \\ R_{magic} : magic(S) \leftarrow magic(Q), Lit_1, \dots, Lit_m$$

R_{magic} exists in the magic-transformed program. Because each literal in the body must either be in Ans_0 or be a base fact, $magic(S)$ will be derived by the SNMT evaluation.

Figure 4: Part of a tabled evaluation

- Inductive Case. Assume the statements 1 and 2 hold at all iterations less than N to demonstrate that the statements will also hold at N .
 - 1) *Non-magic Facts*: In the case of an answer A derived by the Breadth-First Scheduling evaluation, the rule

$$R : Head \leftarrow Lit_1, \dots, Lit_N$$

was used to derive A through a program clause resolution step $PCR(Q, R)$ followed by a series of resolution steps of answers against selected literals of consuming nodes. It must be the case that each of the body literals is either a base fact, or is in Ans_{N-1} . It remains to show that at least one of the answers is contained in δAns_{N-1} , or that the subgoal Q is in $\delta Subgoals_{N-1}$. To see this, suppose that in the forest for the Breadth-First Scheduling evaluation, A is in tree T , and consider the path $C = C_1, \dots, C_n$ of consuming nodes leading from the root of T to the particular instance of A derived at N . (Note that there could be more than one such answer leaf A in T . We need consider only a particular instance produced at iteration N). This situation is depicted in Figure 4. For the purpose of simplicity, first assume that $SC(Q)$ was added before iteration $N-1$. In this case, the path C cannot be empty — R cannot be a bodyless clause — otherwise this instance of A would have been added before iteration $N-1$. Therefore, there must be at least a consuming node and the path cannot be empty. Consider the actions, in iteration N , of resolving an answer against some consuming node, say C_i , in the path. This resolution produces a new consuming node, C_{i+1} , and any answers to the subgoal of C_{i+1} that were produced in previous iterations are resolved against C_{i+1} in iteration N (cf. the *add_operation_{BreadthFirst}* for ACR_{cns} in Definition 4.4). Extending this argument, it is easy to see that the rest of the path from C_{i+1} to A is created at iteration N . Thus, if Q is not in $\delta Subgoals_{N-1}$ it must be the case that some answer used for resolution along the path C must be in δAns_{N-1} . The argument can be easily extended to demonstrate that if no answer used along C is in δAns_{N-1} , then Q must be in $\delta Subgoals_{N-1}$.

This argument shows that in the Breadth-First Scheduling evaluation, an answer leaf A is produced iff its subgoal is in the delta set, or one of the consuming nodes resolves against an answer in the delta set. Now

consider the SNMT evaluation. By rewriting rule 2 of Definition B.4, a similar rule to the one above is used by SNMT:

$$R : Q \leftarrow Lit_1, \dots, Lit_N \quad \Rightarrow$$

$$R_{magic} : Q \leftarrow magic(Q), Lit_1, \dots, Lit_N$$

By the induction hypothesis, any derived literals in Ans_N in the Breadth-First Scheduling evaluation are also in the SNMT evaluation (Q must have been derived), and by the properties of SNMT, at least one of the derived or magic facts is in the delta set for SNMT. Thus SNMT will produce A at iteration N .

- 2) *Magic Fact*: Suppose the Breadth-First Scheduling evaluation adds $SC(Q)$ a scheduling sequence σ_k . In this case, the Breadth-First Scheduling evaluation has used for resolution a rule for a subgoal Q' whose body consists of a prefix, Lit_1, \dots, Lit_m leading a call to a rule whose head unifies with S . By rewrite rule 3 of Definition B.4

$$R : Q' \leftarrow Lit_1, \dots, Lit_m, S \quad \Rightarrow$$

$$R_{magic} : magic(S) \leftarrow magic(Q'), Lit_1, \dots, Lit_m$$

R_{magic} exists in the magic-transformed program. By an argument essentially the same as for non-magic facts, for the Breadth-First Scheduling evaluation, either Q' is in $\delta Subgoals_{N-1}$ or at least one answer of the Breadth-First Scheduling evaluation must be in δAns_{N-1} . In addition, any answers used for resolution on the path of consuming nodes to S must be in Ans_{N-1} . By the induction hypothesis, SNMT would contain the same derived and magic answers in the delta sets for the same iteration. Through the above rule it also produces $magic(S)$ in the SNMT evaluation.

◇