The Limits of Fixed-Order Computation

Konstantinos Sagonas, Terrance Swift, David S. Warren

Department of Computer Science
State University of New York at Stony Brook
Stony Brook, NY 11794-4400
{kostis,tswift,warren}@cs.sunysb.edu

Abstract. Fixed-order computation rules, used by Prolog and most deductive database systems, do not suffice to compute the well-founded semantics [19] because they cannot properly resolve loops through negation. This inadequacy is reflected both in formulations of SLS-resolution [8, 12] which is an ideal search strategy, and in more practical strategies like SLG [3], or Well-Founded Ordered Search [16]. Typically, these practical strategies combine an inexpensive fixed-order search with a relatively expensive dynamic search, such as an alternating fixpoint [18]. Restricting the search space of evaluation strategies by maximizing the use of fixed-order computation is of prime importance for efficient goal-directed evaluation of the well-founded semantics.
Towards this end, the theory of *modular stratification* [13], formulates a subset of normal logic programs whose literals can be statically reordered so that the program can be evaluated using a fixed-order computation rule. However, exploration of larger classes of stratified programs that can be evaluated in this manner has been left open in the literature, perhaps due to the lack of implementation methods that can benefit from such results. We address the limits of fixed-order computation by adapting results of Przymusinski [8] to formulate the class of *left-to-right dynamically stratified programs*. We show that this class properly includes other classes of fixed-order stratified programs. Furthermore, we introduce SLG_{strat}, a variant of SLG resolution that uses a fixed-order computation rule, and prove that it correctly evaluates ground left-to-right dynamically stratified programs. We outline how SLG_{strat} can be used as a basis for restricting the space of possible SLG derivations and, finally, outline how these results are used in the abstract machine of XSB, and can be used in other methods such as Ordered Search of CORAL.

1 Introduction

It is known that the well-founded semantics (WFS) combines the features of both the *stable model* [5] and *partial Clark completion* [9] semantics and thus ensures a proper behavior under both positive and negative recursion. Also, it can be characterized as the only semantics which is both *rational* and *cumulative* [4]. As a result, the WFS provides a natural and intuitive semantics for the class of *all* normal logic programs. Furthermore, the WFS can be computed with polynomial data complexity for function-free programs, and as a result several practical evaluation strategies have been formulated for it. Despite all these properties, the WFS cannot be computed using a fixed computation rule. The reason is that an evaluation of a normal program may encounter a cycle through negation and mechanisms are needed to avoid or to break such cycles. The standard procedural semantics for the WFS is SLS-resolution [8, 12]. Unfortunately, SLS-resolution is not effective in general. As formulated by Przymusinski [8] SLS-resolution uses an oracle to select literals from lower dynamic strata, while Global SLS-resolution as formulated by Ross [12] requires a positivistic computation rule and that all negative literals are ideally evaluated in parallel. These departures from fixed computation are naturally reflected in more practical strategies for evaluating the WFS of normal logic programs [2, 3, 6, 16]. Typically, these strategies consist mainly of a phase that uses a fixed computation rule, followed by some other phase — usually more expensive — which, if necessary, alters that computation rule in order to try to avoid or break recursion through negation. In the case of the SLG resolution strategy of [3], tabling is the basis of the first of these phases while delay and simplification steps are used to break cycles through negation. In the Well-Founded Ordered Search of [16], magic templates with Ordered Search [10] are used for the first phase, while an alternating fixpoint similar to that of [18] is used for the second. The limitations of fixed-order computation[1] can be seen clearly in the following example.

Example 1 [13]. Consider the following program and query `p(a)`.

```
p(X) :- t(X,Y,Z), ¬p(Y), ¬p(Z).      t(a,b,a).
p(b).                                t(a,a,b).
```

The rule instance `p(a) :- t(a,Y,Z),¬p(Y),¬p(Z).` cannot be evaluated using any fixed computation rule. After `t(a,b,a)` has been resolved against the first literal of that rule instance, the second literal must be chosen before the third to avoid the negative dependency through `p(a)`; after `t(a,a,b)` has been used for resolution, the second literal should be skipped and the third literal must be resolved before it in order to properly fail for `p(a)`. ■

Fixed computation rules are important for efficient implementations of systems based on logic; for example most database optimizations rely on the presence of a fixed computation rule. More importantly, however, a break from a fixed computation rule subtly undermines the goal-orientation of techniques like tabling or magic templates since it may unnecessarily open up the search space by introducing subgoals that need not be *relevant* to proving a query. As a matter of fact, even the notion of goal-directedness is probably ill-defined in the absence of a fixed computation rule.

Given the importance of a fixed computation rule, it is natural to try to determine what its limits are and in what cases an evaluation has to break the fixed-order computation. The most notable attempt to address this problem is through the formalism of *modular stratification.* The salient feature of the modularly stratified class of programs, from the perspective of fixed computation rules, is that if a program is modularly stratified, then it is statically reorderable

[1] Without loss of generality, a fixed left-to-right computation rule can be used to exemplify the problems of all fixed computation rules, and so we will equate the left-to-right computation rule with general fixed rules throughout the paper. All results can be easily extended to any fixed computation rule.

so that it is modularly stratified for a fixed computation rule. This property has made modular stratification useful for deductive database systems based on magic-set evaluation (e.g. [11]). However, whether a program is modularly stratifiable at all is undecidable. Furthermore, programs which can be evaluated using a fixed computation rule may not be modularly stratified, as the program in the following example.

Example 2. The following program is neither *modularly*, nor *weakly stratified* [7].

```
s :- ¬s, p.              p :- q, ¬r, ¬s.
s :- ¬p, ¬q, ¬r.         q :- r, ¬p.
                         r :- p, ¬q.
```

■

To our knowledge, the only stratification class that includes the program in Example 2 is *dynamic stratification* of [8] (also independently introduced by Bidoit and Froidevaux as *effective stratification* in [1]). From the perspective of the well-founded semantics, dynamic stratification is extremely powerful since all normal logic programs with two-valued well-founded models are dynamically stratified. The reason for the power of dynamic stratification is that components used in standard stratifications are determined statically, based on the syntactic form of the program, and more precisely, on the predicate dependency graph of the program. Therefore, these *components* do not depend in any way on the truth or falsity of the atoms appearing in the graph. This makes them very simple to use, but at the same time, limits their applicability to relatively narrow classes of programs. On the other hand, components of (left-to-right) dynamic stratification — as the name indicates — are generated dynamically. This process eliminates atom dependencies that can never be satisfied. Given the power of dynamic stratification, the goal of this paper is to study a restriction of this class to a fixed-order computation rule. We believe this restriction gives insight into the limits of fixed-order computations in general, and indicates useful features for effective evaluation methods for the well-founded semantics.

More specifically, the contributions of this paper are as follows.

1. We define a subclass of dynamically stratified programs, called *left-to-right dynamically stratified* (*LRD-stratified*) programs, and show that this class properly includes the class of left-to-right weakly stratified programs, and effectively includes all modularly stratified programs.
2. We introduce a variant of SLG resolution, called SLG_{strat}, which uses a fixed computation rule to evaluate ground LRD-stratified programs. SLG_{strat} differs from SLG in that:
   (a) It suspends negative literals rather than delaying them.
   (b) It introduces the novel technique of *early completion* which uses information from *within* a dynamic stratum to break and sometimes avoid cycles through negation.

The structure of the rest of the paper is as follows. We define LRD-stratified programs in Section 2. We then introduce SLG_{strat} in Section 3, and in Section 4 show the completeness of SLG_{strat} for ground LRD-stratified programs. Section 5 addresses the issue of relevance and goal-orientation of evaluation strategies and sketches how SLG_{strat} can be used as a basis to constrain the search space of possible derivations of full SLG. We conclude with a discussion of difficulties involved in formulating techniques for the open problem of evaluating non-ground LRD-stratified programs, and how these results are potentially applicable to other evaluation methods for stratified negation.

2 Left-to-Right Dynamically Stratified Programs

We assume the standard terminology of logic programming. The *Herbrand base* H_P of a program P is the set of all ground atoms. By a *3-valued interpretation* I of a program P we mean a pair $\langle T; F\rangle$, where T and F are subsets of the Herbrand base H_P of P. The set T contains all ground atoms that are true in I, the set F contains all ground atoms that are false in I, and the truth value of the remaining atoms in $U = H_P - (T \cup F)$ is undefined. We usually require that the interpretation I is *consistent*; i.e. that sets T and F are disjoint. If A is a ground atom from H_P then we write $val_I(A) = \mathtt{t}$ ($val_I(A) = \mathtt{f}$ or $val_I(A) = \mathtt{u}$) if A is true (false or undefined) in I, respectively. We call $val_I(A)$ the *truth value* of A in I. A 3-valued interpretation of P is called a 2-valued interpretation of P if all ground atoms are either true or false in I, i.e. if $H_P = T \cup F$. A consistent 3-valued interpretation M is a *3-valued model* of P if $val_M(C) = \mathtt{t}$, for every clause C in P. If M is 2-valued then it is called a *2-valued* or a *total model* of P. Any consistent 3-valued interpretation can be viewed as a function from the Herbrand base H_P to the three-element set $\{\mathtt{f}, \mathtt{u}, \mathtt{t}\}$, ordered by $\mathtt{f} < \mathtt{u} < \mathtt{t}$.

2.1 Iterated Fixed Point under a Left-to-Right Computation Rule

We first adapt Przymusinski's definition of the iterated fixed point from [8] to a non-ideal left-to-right computation rule. Our main modification is the introduction of the notion of a *failing prefix* in defining the iterated fixed point operators $\mathcal{T}_I$ and $\mathcal{F}_I$ (Definition 1). Intuitively, I represents facts currently known to be true or false and $\mathcal{T}_I(T)$ ($\mathcal{F}_I(F)$) contains facts not contained in I, whose truth (falsity) can be immediately derived from P under the interpretation I, and assuming that all facts in T are true (all facts in F are false). The failing prefix constraint prevents false facts from inclusion in $\mathcal{F}_I(F)$, unless their falsity can be established by a left-to-right examination of the literals in clauses of P.

Definition 1. For sets T and F of ground atoms we define

$\mathcal{T}_I(T) = \{A : val_I(A) \neq \mathtt{t}$ and there is a clause $B \leftarrow L_1, ..., L_n$ in P and a ground substitution θ such that $A = B\theta$ and for every $1 \leq i \leq n$ either $L_i\theta$ is true in I, or $L_i\theta \in T\}$;

$\mathcal{F}_I(F) = \{A : val_I(A) \neq \mathtt{f}$ and for every clause $B \leftarrow L_1, ..., L_n$ in P and ground substitution θ such that $A = B\theta$ there exists a *failing prefix*: i.e. there is some i $(1 \leq i \leq n)$, such that $L_i\theta$ is false in I or $L_i\theta \in F$, and for all j $(1 \leq j \leq i-1)$, $L_j\theta$ is true in $I\}$. □

Even though the conditions $val_I(A) \neq \mathtt{t}$ and $val_I(A) \neq \mathtt{f}$ are inessential, they ensure that only *new* facts are included in $\mathcal{T}_I(T)$ and $\mathcal{F}_I(F)$, and will simplify the definition of dynamic strata below. Subsets T_I and F_I of the Herbrand base are obtained by iterating the operators $\mathcal{T}_I$ and $\mathcal{F}_I$ according to the following definition.

Definition 2. **[8]** Let $I = \langle T; F \rangle$ be an interpretation, we define

$$T_I^{\uparrow 0} = \emptyset \text{ and } F_I^{\downarrow 0} = H_P$$
$$T_I^{\uparrow n+1} = \mathcal{T}_I(T_I^{\uparrow n}) \text{ and } F_I^{\downarrow n+1} = \mathcal{F}_I(F_I^{\downarrow n})$$
$$T_I = \bigcup_{n<\omega} T_I^{\uparrow n} \text{ and } F_I = \bigcap_{n<\omega} F_I^{\downarrow n}$$

Proposition 3. **[8]** *The set T_I is the least fixed point of the operator $\mathcal{T}_I$, and the set F_I is the least fixed point of the operator $\mathcal{F}_I$.*

Definition 4. **[8]** Let $\mathcal{I}$ be the operator which assigns to every interpretation I of P a new interpretation $\mathcal{I}(I)$ defined by: $\mathcal{I}(I) = I \cup \langle T_I; F_I \rangle$. □

The operator $\mathcal{I}$ extends the interpretation I to $\mathcal{I}(I)$ by adding to I new atomic facts T_I which can be derived from P knowing I, and the negations of new atomic facts F_I which can be assumed false in P just by knowing I.

Definition 5 Iterated Fixed Point. **[8]** Let:

$$J_0 = \langle \emptyset; \emptyset \rangle;$$
$$J_{\alpha+1} = \mathcal{I}(J_\alpha), \text{ i.e. } J_{\alpha+1} = J_\alpha \cup \langle T_{J_\alpha}; F_{J_\alpha} \rangle;$$
$$J_\alpha = \bigcup_{\beta<\alpha} J_\beta, \text{ for limit } \alpha.$$

Let $J_{IF(P)}$ denote the fixed point interpretation J_δ, where δ is the smallest countable ordinal such that both sets T_{J_δ} and F_{J_δ} are empty. We refer to δ as the *depth* of program P. □

$J_{IF(P)}$ is a consistent 3-valued interpretation of P. It can be proven that $J_{IF(P)}$ is the least fixed point of the operator $\mathcal{I}$. Note that $J_{IF(P)}$ is an informationally sound approximation of the well-founded model of P. Furthermore, as will soon be proven, $J_{IF(P)}$ is the well-founded model of P if P is left-to-right dynamically stratified. We introduce this class of programs in the following section.

2.2 Left-to-Right Dynamic Stratification

As observed in [8], the iterated fixed point induces a natural stratification on a program P.

Definition 6 *Dynamic Stratum under a Fixed-Order Computation Rule.* Let α be a countable ordinal. The *α's dynamic stratum* S_α of P under a left-to-right computation rule is defined as

$$S_\alpha = \{A : A \in (T_{J_\alpha} \cup F_{J_\alpha})\}$$

for any α such that $T_{J_\alpha} \cup F_{J_\alpha}$ is not empty. Let δ be the minimal ordinal such that for all non-empty strata, S_α, $\alpha < \delta$; then we call δ the *ultimate stratum* under a left-to-right computation rule, and

$$S_\delta = \{A : A \in H_P - \bigcup_{\alpha<\delta} S_\alpha\}$$

Conversely, let A be a ground atom. The *dynamic stratum* of A under a left-to-right computation rule is the unique ordinal number α such that $A \in (T_{J_\alpha} \cup F_{J_\alpha})$, and is δ otherwise. In general, for any possibly non-ground atom A the *dynamic stratum* of A is equal to:

$$stratum(A) = sup\{stratum(A\theta) : \theta \text{ is a ground substitution}\}$$

and let $str(\neg A) = min(\delta, str(A) + 1)$. □

The dynamic stratification of P under a left-to-right computation rule is therefore a decomposition of the set of all atoms in H_P into disjoint strata. The α's dynamic stratum of P is defined at level $\alpha < \delta$ as the set of all atoms that were *newly added* to the interpretation J_α in order to obtain $J_{\alpha+1}$ (i.e. those atoms whose truth or falsity was determined at this very level). The ultimate stratum S_δ contains all the remaining atoms, i.e. all atoms whose truth value could not be determined when the fixed point was reached using a left-to-right computation rule. Some of the atoms in the ultimate stratum may be undefined in the well-founded model M_P of P, or their truth value could be determined by using a non-fixed (possibly ideal) computation rule.

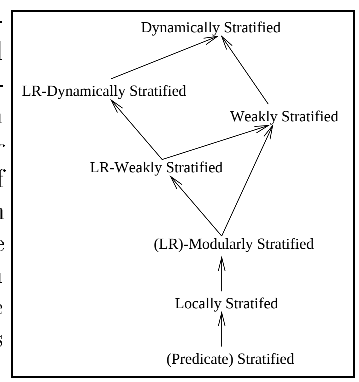
Definition 7 Left-to-Right Dynamically Stratified Program. A program P is *left-to-right dynamically stratified* if its ultimate stratum S_δ is empty. □

Theorem 8. *Let P be a left-to-right dynamically stratified program. Then the interpretation $J_{IF(P)}$ is a minimal 2-valued model of P.*

Since $J_{IF(P)}$ is the least fixed point of the operator $\mathcal{I}$, which itself is defined using least fixed points of the operators $\mathcal{T}_I$ and $\mathcal{F}_I$, $J_{IF(P)}$ is an iterated fixed point model of P. It also coincides with the well-founded [19] and the unique stable model [5] of the program P.

2.3 Relationships between stratification classes

The power of dynamic stratification and of its restriction to dynamic stratification under a fixed computation rule, arises from the fact that components are determined dynamically rather than statically as they are in *weak* [7], or in *modular stratification* [13]. Another reason for the power of dynamic stratification is that it uses information from *within* the (dynamic) stratum to eliminate atom dependencies that can never be satisfied. In fact, a hierarchy of stratification classes can be defined as in the figure on the left where arrows indicate inclusion.



Most of the relationships between the stratification classes presented in the above figure are well-known in the literature. Theorem 9 substantiates the placement of LRD-stratification within the stratification hierarchy. The proof of Theorem 9, along with supporting definitions of the class of *left-to-right weakly stratified programs*, can be found in the full version of the paper.

Theorem 9. *The class of LRD-stratified programs properly includes the class of left-to-right weakly stratified, and (left-to-right) modularly stratified programs.*

3 Tabled Evaluation of LRD-stratified Programs

This section presents the formal definitions of SLG_{strat}, a resolution strategy based on tabling which will be shown to evaluate LRD-stratified programs using a left-to-right fixed-order computation rule. SLG_{strat} is largely a restriction of SLG [3] and inherits most of its properties. Differences of SLG_{strat} from SLG are noted as they occur. In particular, SLG_{strat} adds a NEGATIVE SUSPENSION transformation and its necessary underpinnings along with an altered definition of when a set of subgoals is completely evaluated. The latter and the introduction of the NEGATIVE SUSPENSION transformation require a change of the COMPLETION transformation so that suspended computations are resumed.

3.1 An Example

To motivate the following sections that contain technical definitions of SLG_{strat}, we present the SLG_{strat} evaluation of a query `?- q(a).` against the following LRD-stratified program. Each program clause has been labeled for convenience of reference.

C_1 :	`q(X) :- t(X), ¬q(X).`	C_4 :	`p :- t(a), r.`
C_2 :	`q(a) :- ¬p, ¬r.`	C_5 :	`r :- p, ¬r, ¬q.`
C_3 :	`t(a).`	C_6 :	`q :- q(X).`

A *system* is a set of pairs of the form $(A : \rho)$, where A is a subgoal and ρ is a sequence of annotated rules for A. No two pairs in a system have the same (variant) subgoal. The sequence ρ of annotated rules represents the history of the addition, suspension, and/or deletion of rules for subgoal A. The annotation of each rule in ρ indicates where the rule is derived from and whether some rule is suspended or disposed. Intuitively, a rule is disposed when it no longer has anything to contribute to the derivation. In general, the sequence ρ can be transfinite.

The initial system $\mathcal{S}_0$ is empty. The query atom $q(a)$ is introduced as the first subgoal. Using the NEW SUBGOAL transformation, the initial sequence of $q(a)$ is an arbitrary sequence of all rules obtained by resolving $q(a) \leftarrow q(a)$ on $q(a)$ in the body with program clauses. After $q(a)$ is introduced the system is:

$$\mathcal{S}_1 = \left\{ q(a) : \begin{bmatrix} 0 : \mathtt{q(a)} \leftarrow\ \mathtt{t(a)},\ \neg\mathtt{q(a)}. & \langle C_1 \rangle \\ 1 : \mathtt{q(a)} \leftarrow\ \neg\mathtt{p},\ \neg\mathtt{r}. & \langle C_2 \rangle \end{bmatrix} \right\}$$

An annotated rule of the form $G\langle C \rangle$, where C is (the label of) a program clause, means that G is obtained by resolution with program clause C. The sequence of a subgoal is written as an array of pairs $\alpha : AG$ where α is an ordinal and AG is an annotated rule. In general, the annotation in AG indicates where the rule in AG is derived from — either a program clause or an earlier rule in the sequence, and in the latter case, whether the earlier rule in the sequence is disposed. For each non-disposed rule in the sequence of a subgoal, its head captures variable bindings that have been accumulated, and its body contains literals that remain to be solved in order to derive answers for the subgoal. For a rule with a non-empty body, a literal is selected from the rule body by a computation rule. The atom of the selected literal is added as a new subgoal if it is not in the current system. In the example above, `t(a)` is selected from the body of `q(a)` $\leftarrow$ `t(a), ` $\neg$`q(a)` and is added as a new subgoal. Like $q(a)$, $t(a)$ has its own sequence of annotated rules obtained by resolution of $t(a) \leftarrow t(a)$ on $t(a)$ in the body with program clauses. So the system becomes:

$$\mathcal{S}_2 = \left\{ q(a) : \begin{bmatrix} 0 : \mathtt{q(a)} \leftarrow \mathtt{t(a)},\ \neg\mathtt{q(a)}. & \langle C_1 \rangle \\ 1 : \mathtt{q(a)} \leftarrow \neg\mathtt{p},\ \neg\mathtt{r}. & \langle C_2 \rangle \end{bmatrix} \ t(a) : \begin{bmatrix} 0 : \mathtt{t(a)}. \ \langle C_3 \rangle \end{bmatrix} \right\}$$

A rule with an empty body is an *answer*, e.g., `t(a)` in the above sequence for $t(a)$. Note that this subgoal has obtained all answers it will ever have; any other rules no longer have anything to contribute to the derivation. Subgoals that get their most general answer can be completed without taking into account possible dependencies that they might have. We call such subgoals *early completable* and the operation that is then performed *early completion* of subgoals. Early completion is a special case where a set of subgoals is completely evaluated and the COMPLETION transformation is applicable (see Definition 18).

Answers are returned to the appropriate selected literals through the POSITIVE RETURN transformation. By returning the newly derived answer, the sequence of $q(a)$ is extended as follows:

$$q(a) : \begin{bmatrix} 0 : \mathtt{q(a)} \leftarrow \mathtt{t(a)},\ \neg\mathtt{q(a)}. & \langle C_1 \rangle \\ 1 : \mathtt{q(a)} \leftarrow \neg\mathtt{p},\ \neg\mathtt{r}. & \langle C_2 \rangle \\ 2 : \mathtt{q(a)} \leftarrow \neg\mathtt{q(a)}. & \langle 0, t(a) \rangle \end{bmatrix}$$

This last annotation indicates that `q(a)` $\leftarrow$ $\neg$`q(a)` is obtained from the rule corresponding to ordinal 0 by solving the selected atom using an answer $t(a)$. The selected literal of this sequence is negative. Its subgoal does not have any answers, and it is not yet known whether it will ever get any. Consequently, the selected negative literal cannot be resolved at this point. SLG_{strat} *suspends* negative selected literals whose truth value is not known. This is done by employing the NEGATIVE SUSPENSION transformation. Doing so, the sequence of $q(a)$ is extended in two steps to that shown in system $\mathcal{S}_9$ below.

Program clause resolution is then used to derive answers for p which is the subgoal of the selected literal of the second suspended rule. After a sequence of NEW SUBGOAL, POSITIVE RETURN, and NEW SUBGOAL transformations the system is as follows:

$$\mathcal{S}_9 = \left\{ q(a) : \begin{bmatrix} 0 : \mathtt{q(a)} \leftarrow \mathtt{t(a)},\ \neg\mathtt{q(a)}. & \langle C_1 \rangle \\ 1 : \mathtt{q(a)} \leftarrow \neg\mathtt{p},\ \neg\mathtt{r}. & \langle C_2 \rangle \\ 2 : \mathtt{q(a)} \leftarrow \neg\mathtt{q(a)}. & \langle 0, t(a) \rangle \\ 3 : \mathit{suspended} & \langle 2 \rangle \\ 4 : \mathit{suspended} & \langle 1 \rangle \end{bmatrix} \begin{array}{r} t(a) : \begin{bmatrix} 0 : \mathtt{t(a)}. \ \langle C_3 \rangle \end{bmatrix} \\ p : \begin{bmatrix} 0 : \mathtt{p} \leftarrow \mathtt{t(a)}, \mathtt{r}. & \langle C_4 \rangle \\ 1 : \mathtt{p} \leftarrow \mathtt{r}. & \langle 0, t(a) \rangle \end{bmatrix} \\ r : \begin{bmatrix} 0 : \mathtt{r} \leftarrow \mathtt{p},\ \neg\mathtt{r},\ \neg\mathtt{q}. \ \langle C_5 \rangle \end{bmatrix} \end{array} \right\}$$

Note that subgoals p and r form an *unfounded set* [19] and cannot get any answers. But they are not completed since they have rules in their sequences which have a selected literal and are not disposed. On the other hand, the literal ¬`p` cannot be unsuspended until the truth value of its subgoal is known. SLG_{strat} handles this situation through the COMPLETION transformation that disposes all rules (that are not answers) of a set of mutually dependent subgoals when these subgoals have performed all possible operations. Applying the COMPLETION transformation to the set $\{p, r\}$ disposes all rules of these subgoals, and unsuspends rules of the system that are suspended on the completion of p or r. The resulting system is the following:

$$\mathcal{S}_{10} = \left\{ \begin{array}{ll} q(a) : \left[\begin{array}{ll} 0 : \mathtt{q(a)} \leftarrow \mathtt{t(a)},\ \neg\mathtt{q(a)}. & \langle C_1 \rangle \\ 1 : \mathtt{q(a)} \leftarrow \neg\mathtt{p},\ \neg\mathtt{r}. & \langle C_2 \rangle \\ 2 : \mathtt{q(a)} \leftarrow \neg\mathtt{q(a)}. & \langle 0, t(a) \rangle \\ 3 : suspended & \langle 2 \rangle \\ 4 : suspended & \langle 1 \rangle \\ 5 : unsuspended & \langle 1 \rangle \end{array} \right] & \begin{array}{l} t(a) : \left[0 : \mathtt{t(a)}.\ \langle C_3 \rangle \right] \\ p : \left[\begin{array}{ll} 0 : \mathtt{p} \leftarrow \mathtt{t(a)}, \mathtt{r}. & \langle C_4 \rangle \\ 1 : \mathtt{p} \leftarrow \mathtt{r}. & \langle 0, t(a) \rangle \\ 2 : disposed & \langle 1 \rangle \end{array} \right] \\ r : \left[\begin{array}{ll} 0 : \mathtt{r} \leftarrow \mathtt{p},\ \neg\mathtt{r},\ \neg\mathtt{q}. & \langle C_5 \rangle \\ 1 : disposed & \langle 0 \rangle \end{array} \right] \end{array} \end{array} \right\}$$

Subgoals p and r are completed without any answers; by negation as failure literals ¬`p` and ¬`r` can be removed using two NEGATIVE RETURN transformations. Note that since the value of subgoal r is known, a NEGATIVE SUSPENSION transformation is not needed. Like $t(a)$, subgoal $q(a)$ also got its most general answer. As a result, it is early completable. A COMPLETION transformation on $q(a)$ takes place and the resulting final system is as follows:

$$\mathcal{S}_{13} = \left\{ \begin{array}{ll} q(a) : \left[\begin{array}{ll} 0 : \mathtt{q(a)} \leftarrow \mathtt{t(a)},\ \neg\mathtt{q(a)}. & \langle C_1 \rangle \\ 1 : \mathtt{q(a)} \leftarrow \neg\mathtt{p},\ \neg\mathtt{r}. & \langle C_2 \rangle \\ 2 : \mathtt{q(a)} \leftarrow \neg\mathtt{q(a)}. & \langle 0, t(a) \rangle \\ 3 : suspended & \langle 2 \rangle \\ 4 : suspended & \langle 1 \rangle \\ 5 : unsuspended & \langle 1 \rangle \\ 6 : \mathtt{q(a)} \leftarrow \neg\mathtt{r}. & \langle 1 \rangle \\ 7 : \mathtt{q(a)} & \langle 6 \rangle \\ 8 : disposed & \langle 2 \rangle \end{array} \right] & \begin{array}{l} t(a) : \left[0 : \mathtt{t(a)}.\ \langle C_3 \rangle \right] \\ p : \left[\begin{array}{ll} 0 : \mathtt{p} \leftarrow \mathtt{t(a)}, \mathtt{r}. & \langle C_4 \rangle \\ 1 : \mathtt{p} \leftarrow \mathtt{r}. & \langle 0, t(a) \rangle \\ 2 : disposed & \langle 1 \rangle \end{array} \right] \\ r : \left[\begin{array}{ll} 0 : \mathtt{r} \leftarrow \mathtt{p},\ \neg\mathtt{r},\ \neg\mathtt{q}. & \langle C_5 \rangle \\ 1 : disposed & \langle 0 \rangle \end{array} \right] \end{array} \end{array} \right\}$$

Note that all the subgoals of the final system are completed, and that the use of the NEGATIVE SUSPENSION transformation in conjunction with the *early completion* operation — which takes into account the truth value of subgoals in the system — avoided the loop through negation involving subgoal $q(a)$.

3.2 SLG Systems

A *subgoal* is an atom. In our tabling framework, two subgoals are considered the same if they are identical up to variable renaming. A *literal* is an atom, or the negation of an atom A denoted as $\neg A$.

SLG evaluations are modeled through sequences of *X-rules* associated with subgoals encountered in a system. The set of subgoals with their sequences of X-rules intuitively corresponds to the forest of trees representation commonly used by tabling methods [2].

Definition 10 [3]. An *X-rule* C is of the form:

$$H \leftarrow L_1, \ldots, L_n.$$

where H is an atom, each $L_i (1 \leq i \leq n)$ is a literal, and $n \geq 0$. If $n = 0$, C is called an *answer*.

A *computation rule* is an algorithm that selects from the body of an X-rule C a literal L. L is called the *selected literal* of C. If L is an atom, it is also called the *selected atom* of C. □

X-rules are used to form X-elements which comprise the above-mentioned sequences.

Definition 11 X-element. Let P be a finite program, C be a rule in P, G be an X-rule, α be an ordinal, and H be an atom. Then an *X-element* is of the form $G\langle C\rangle$, $G\langle\alpha\rangle$, $G\langle\alpha, H\rangle$, $disposed\langle\alpha\rangle$, $suspended\langle\alpha\rangle$, or $unsuspended\langle\alpha\rangle$. □

Definition 12 X-sequence. **[3]** An *X-sequence* ρ is a mapping from all ordinals that are smaller than some ordinal α to the set of X-elements. The ordinal α is the *length* of ρ. □

Each subgoal in a system has an associated X-sequence. The X-sequence captures the history of the addition/disposal of X-rules and, in SLG_{strat}, the suspension/unsuspension of X-rules for the subgoal during its partial deduction. Each X-element in the X-sequence indicates the origin of an X-rule — a rule in a program or an X-rule earlier in the X-sequence — along with whether an X-rule is disposed, suspended, or unsuspended.

More specifically, let ρ be the X-sequence of a subgoal A. Let β be an ordinal such that $\rho(\beta) = e$ for some X-element e, and α be an ordinal such that $\alpha < \beta$.

- If e is of the form $G\langle C\rangle$, where G is an X-rule and C is a rule in a program, then G is created by resolving $A \leftarrow A$ on A in the body with C;
- If e is of the form $G\langle\alpha\rangle$, where G is an X-rule, then the X-rule corresponding to α in ρ is disposed and replaced by G;
- If e is of the form $G\langle\alpha, H\rangle$, where G is an X-rule and H is an atom, then G is obtained by solving the selected atom of the X-rule corresponding to α in ρ using an answer H;
- If e is $disposed\langle\alpha\rangle$, then the X-rule corresponding to α in ρ is simply disposed.
- If e is $suspended\langle\alpha\rangle$, then the X-rule corresponding to α in ρ is marked as suspended.
- If e is $unsuspended\langle\alpha\rangle$, then the X-rule corresponding to α in ρ is marked as unsuspended.

Intuitively, X-rules that are disposed from an X-sequence of a system do not appear in any future point in the sequence. Similarly, X-rules of a subgoal that are marked as suspended do not participate in the evaluation of the subgoal until they are marked as unsuspended.

We consider two main operations over X-sequences: concatenation and the least upper bound of an increasing chain of X-sequences. The former is used to extend the X-sequence of a subgoal, and the latter is needed for the transfinite definition of an SLG_{strat} derivation that will be presented in Section 3.4.

Definition 13. Let ρ_1 and ρ_2 be X-sequences of length α_1 and α_2 respectively. The *concatenation* of ρ_1 with ρ_2, denoted by $\rho_1 \cdot \rho_2$, is an X-sequence of length $\alpha_1 + \alpha_2$ such that $(\rho_1 \cdot \rho_2)(i) = \rho_1(i)$ for every $i (0 \leq i < \alpha_1)$, and $(\rho_1 \cdot \rho_2)(\alpha_1 + j) = \rho_2(j)$ for every $j (0 \leq j < \alpha_2)$.

The X-sequence ρ_1 is said to be a *prefix* of $\rho_1 \cdot \rho_2$. If ρ_2 is a sequence of length 1 such that $\rho_2(0) = e$ for some X-element e, we also write $\rho_1 \cdot \rho_2$ as $\rho_1 \cdot e$. $\square$

There is a natural prefix partial order over X-sequences. Let ρ_1 and ρ_2 be X-sequences. Then $\rho_1 \subseteq \rho_2$ if ρ_1 is a prefix of ρ_2. An X-sequence ρ of length α can also be viewed as a set of pairs $\{\langle i, \rho(i)\rangle \mid 0 \leq i < \alpha\}$, in which case the prefix relation reduces to the subset relation.

Definition 14. Let β be an ordinal and $\rho_i(0 \leq i < \beta)$ be an increasing chain of X-sequences (with respect to $\subseteq$), and let α be an ordinal such that the length of each $\rho_i(0 \leq i < \beta)$ is less than α. Then the least upper bound of the chain $\rho_i(0 \leq i < \beta)$, denoted by $\bigcup\{\rho_i | 0 \leq i < \beta\}$, exists since the length of each $\rho_i(0 \leq i < \beta)$ is less than α. It is the X-sequence that is the union of all $\rho_i(0 \leq i < \beta)$ when an X-sequence is viewed as a set. $\square$

Intermediate states of the evaluation of a query are represented by *systems*.

Definition 15 System. Let P be a program, and R be a computation rule. A *system* $\mathcal{S}$ is a set of pairs of the form $(A : \rho)$, where A is a subgoal and ρ is its X-sequence, such that no two pairs in $\mathcal{S}$ have the same subgoal. A subgoal A is said to be *in* $\mathcal{S}$ if $(A : \rho) \in \mathcal{S}$ for some X-sequence ρ.

Let $(A : \rho) \in \mathcal{S}$, where A is a subgoal and ρ is its X-sequence. Let G be an X-rule and α be an ordinal. G is said to be *the X-rule of A corresponding to α in $\mathcal{S}$* if $\rho(\alpha)$ is either $G\langle C\rangle$, $G\langle i\rangle$, or $G\langle i, H\rangle$, where C is a rule in P, $i < \alpha$, and H is an atom.

Let G be an X-rule of A corresponding to α in $\mathcal{S}$. Then

- G is *disposed* if for some $j > \alpha$, $\rho(j)$ is either $disposed\langle\alpha\rangle$ or is $G'\langle\alpha\rangle$ for some X-rule G';
- G is an *answer of A* if G is not disposed and has no literals in its body;
- G is *suspended* on B if $\neg B$ is the selected literal of G, G is not disposed, and for some $j > \alpha$, $\rho(j)$ is $suspended\langle\alpha\rangle$, and there is no $k > j$ such that $\rho(k)$ is $unsuspended\langle\alpha\rangle$; we say that G is *suspended* if there exists a subgoal B such that G is suspended on B.
- G is an *active X-rule of A* if G is not disposed or suspended and has a selected literal.

A subgoal A in $\mathcal{S}$ is *completed* if all X-rules of A are either answers, or are disposed. $\square$

3.3 Transformations of Systems of Subgoals

Starting with the empty system of subgoals, each transformation transforms one system into another. The initial X-sequence of a subgoal is obtained by resolution with rules in a program. Without loss of generality, we consider only *atomic queries* that contain a single atom.

Definition 16 X-resolution. Let G be an X-rule, of the form $H \leftarrow L_1, \ldots, L_n$, and L_i be the selected atom of G for some $i(1 \leq i \leq n)$. Let C be a rule and C', of the form $H' \leftarrow L'_1, \ldots, L'_m$, be a variant of C with variables renamed so that G and C' have no variables in common. Then G is *X-resolvable* with C if L_i and H' are unifiable. The X-rule:

$$(H \leftarrow L_1, \ldots, L_{i-1}, L'_1, \ldots, L'_m, L_{i+1}, \ldots, L_n)\theta$$

is the *X-resolvent* of G with C, where θ is the mgu of L_i and H'. □

The selected atom in the body of an X-rule can be solved by an answer; this process is called *answer resolution*. The selected negative literal in the body of an X-rule can be solved if the positive counterpart succeeds or fails, according to the following definition.

Definition 17. Let $\mathcal{S}$ be a system. A subgoal A in $\mathcal{S}$ *succeeds* if A has an answer of the form $A' \leftarrow$ where A' is a variant of A, and *fails* if A is completed in $\mathcal{S}$ without any answers. If A neither succeeds nor is completed in $\mathcal{S}$, then we say that A is *evolving* in $\mathcal{S}$. □

Notice that a subgoal containing variables succeeds if it has any answer that does not bind any of its variables. As a special case, a ground subgoal succeeds if it has an answer. The COMPLETION transformation requires the notion of a set of subgoals that are completely evaluated so that their active and their suspended X-rules, can be disposed.

Definition 18 Completely Evaluated. Let $\mathcal{S}$ be a system and Σ be a non-empty set of subgoals in $\mathcal{S}$, none of which is completed. Σ is said to be *completely evaluated* iff for every subgoal $A \in \Sigma$, where $(A : \rho) \in \mathcal{S}$ for some X-sequence ρ, either

- A succeeds, or
- A does not contain any suspended X-rule, and for every active X-rule G of A corresponding to some ordinal α in $\mathcal{S}$, there exists an atom A_1 such that:
  - A_1 is the selected atom of G; and
  - A_1 is a subgoal in $\mathcal{S}$ that is either completed or in Σ; and
  - for every atom H that is the head of some answer of A_1 in $\mathcal{S}$, there exists an ordinal $i > \alpha$ and an X-rule G' such that $\rho(i) = G'\langle\alpha, H\rangle$.

For convenience, we call a subgoal A *completely evaluated* iff there exists a set of subgoals Σ that is completely evaluated, and $A \in \Sigma$. □

Note that according to this definition, any set $\{A\}$ consisting of a single subgoal A that succeeds is a completely evaluated set of subgoals. This condition will be referred to as *early completion* of subgoals. Also note that in a completely evaluated set of subgoals, only those subgoals that succeed can contain X-rules that are suspended on the completion of another subgoal.

Let P be a program, R be an arbitrary but fixed computation rule, and Q be an atomic query. The transformations of SLG_{strat} are listed below:

- $\boxed{\text{NEW SUBGOAL}}$ Let A be a subgoal that is not in $\mathcal{S}$ and that satisfies one of the following conditions:
  - A is the initial atomic query Q; or
  - there is an active X-rule of some subgoal in $\mathcal{S}$ whose selected literal is either A or $\neg A$.

  Let $C_0, C_1, \ldots, C_{k-1} (k \geq 0)$ be all rules in program P with which $A \leftarrow A$ is X-resolvable, and $G_i (0 \leq i < k)$ be the X-resolvent of $A \leftarrow A$ with C_i. Let ρ be the X-sequence of length k such that $\rho(i) = G_i\langle C_i\rangle (0 \leq i < k)$. Then

  $$\frac{\mathcal{S}}{\mathcal{S} \cup \{(A : \rho)\}}$$

  Remark: Repeated subgoals are not solved by resolution with program clauses.

– $\boxed{\text{POSITIVE RETURN}}$ Let $(A : \rho) \in \mathcal{S}$, where A is a subgoal and ρ is its X-sequence. Let G be an active X-rule of A corresponding to an ordinal α in $\mathcal{S}$. Let A_1 be the selected atom of G, and let C be an answer of subgoal A_1 in $\mathcal{S}$ that has H in its head. If there is no ordinal $i > \alpha$ such that $\rho(i) = G_1\langle \alpha, H\rangle$ for some X-rule G_1, then

$$\frac{\mathcal{S}}{\mathcal{S} - \{(A : \rho)\} \cup \{(A : \rho \cdot (G_2\langle \alpha, H\rangle))\}}$$

where G_2 is the X-resolvent of G with C.

Remark: A_1 may have multiple answers with the same atom H in the head. Only one of them is used by POSITIVE RETURN to solve the selected atom A_1 in G. So, in particular, redundant answers are not used.

– $\boxed{\text{NEGATIVE RETURN}}$ Let $(A : \rho) \in \mathcal{S}$, where A is a subgoal and ρ is its X-sequence. Let G be an active X-rule of a subgoal A corresponding to an ordinal α in $\mathcal{S}$, and let $\neg A_1$ be the selected literal of G, where A_1 either succeeds or fails. Then

$$\frac{\mathcal{S}}{\mathcal{S} - \{(A : \rho)\} \cup \{(A : \rho \cdot \mathit{disposed}\langle \alpha\rangle)\}} \text{ if } A_1 \text{ succeeds}$$

$$\frac{\mathcal{S}}{\mathcal{S} - \{(A : \rho)\} \cup \{(A : \rho \cdot G'\langle \alpha\rangle)\}} \quad \text{ if } A_1 \text{ fails}$$

where G' is G with the selected literal $\neg A_1$ deleted.

– $\boxed{\text{NEGATIVE SUSPENSION}}$ Let $(A : \rho) \in \mathcal{S}$, where A is a subgoal and ρ is its X-sequence. Let G be an active X-rule of subgoal A corresponding to an ordinal α in $\mathcal{S}$, and let $\neg A_1$ be the selected literal of G such that A_1 is evolving in $\mathcal{S}$. Then

$$\frac{\mathcal{S}}{\mathcal{S} - \{(A : \rho)\} \cup \{(A : \rho \cdot \mathit{suspended}\langle \alpha\rangle)\}}$$

Remark: Negative literals whose positive counterparts either are completed or succeed in $\mathcal{S}$ are never suspended.

– $\boxed{\text{COMPLETION}}$ Let Σ be a non-empty set of subgoals in $\mathcal{S}$ that is completely evaluated, and let Ξ be the set of subgoals in $\mathcal{S}$ containing an X-rule that is suspended on a literal $\neg A_1$ where $A_1 \in \Sigma$. Then

$$\frac{\mathcal{S}}{\begin{array}{c}\text{for every } B \in \Xi, \text{ replace } (B : \varrho) \in \mathcal{S} \text{ with } (B : \varrho \cdot \varrho') \\ \text{and then} \\ \text{for every } A \in \Sigma, \text{ replace } (A : \rho) \in \mathcal{S} \text{ with } (A : \rho \cdot \rho')\end{array}}$$

where

- ϱ' is an arbitrary sequence of X-elements of the form $\mathit{unsuspended}\langle \alpha\rangle$, where α is an ordinal such that the X-rule of B corresponding to α in ϱ is an X-rule that is suspended on a literal $\neg A_1$ where $A_1 \in \Sigma$.
- ρ' is an arbitrary sequence of X-elements of the form $\mathit{disposed}\langle \alpha\rangle$, where α is an ordinal such that the X-rule of A corresponding to α in ρ is an active X-rule.

Remark: COMPLETION does not depend upon any *a priori* stratification ordering of predicates or atoms. Instead, a set of subgoals is inspected and completed dynamically.

3.4 Derivation and SLG_{strat} Resolution

Definition 19. Let P be a finite program, R be an arbitrary but fixed computation rule, and Q be an atomic query. An SLG_{strat} *derivation* for Q (or *evaluation* of Q) is a sequence of systems $\mathcal{S}_0, \mathcal{S}_1, \ldots, \mathcal{S}_\alpha$ such that:

- $\mathcal{S}_0$ is the empty system $\emptyset$;
- for each successor ordinal $\beta + 1 \leq \alpha$, $\mathcal{S}_{\beta+1}$ is obtained from $\mathcal{S}_\beta$ by an application of one of the transformations, namely, NEW SUBGOAL, POSITIVE RETURN, NEGATIVE RETURN, NEGATIVE SUSPENSION, or COMPLETION;
- for each limit ordinal $\beta \leq \alpha$, $\mathcal{S}_\beta$ is such that $(A : \rho) \in \mathcal{S}_\beta$ if A is a subgoal in $\mathcal{S}_i$ for some $i < \beta$ and $\rho = \bigcup\{\rho' | (A : \rho') \in \mathcal{S}_j \text{ and } j < \beta\}$.

If no transformation is applicable to $\mathcal{S}_\alpha$, $\mathcal{S}_\alpha$ is called a *final system* of Q.

SLG_{strat} *resolution* is the process of constructing an SLG_{strat} derivation for a query Q with respect to a finite program P under a computation rule R. □

To show that a final system always exists, we prove that each SLG_{strat} derivation is a monotonically increasing sequence of systems with respect to some partial ordering and each system in an SLG_{strat} derivation is bounded in size by some ordinal.

Definition 20. Let P be a finite program, and $\mathcal{S}_1$ and $\mathcal{S}_2$ be systems. Then $\mathcal{S}_1 \sqsubseteq \mathcal{S}_2$ if for every $(A : \rho_1) \in \mathcal{S}_1$, there exists $(A : \rho_2) \in \mathcal{S}_2$ such that $\rho_1 \subseteq \rho_2$, i.e., ρ_1 is a prefix of ρ_2. □

The following proposition ensures that the addition of the *suspended* and *unsuspended* statuses in Definition 11, and the introduction of the NEGATIVE SUSPENSION transformation do not affect fundamental properties of SLG resolution.

Proposition 21. *Let P be a finite program, R be an arbitrary but fixed computation rule, and Q be an atomic query. Then*

(a) *there exists some ordinal λ such that for any system $\mathcal{S}$ in any SLG_{strat} derivation for Q, the length of the X-sequence of each subgoal in $\mathcal{S}$ is bounded by λ;*

(b) *every SLG_{strat} derivation of Q is a monotonically increasing sequence of systems with respect to $\sqsubseteq$; and*

(c) *there exists some SLG_{strat} derivation for Q, $\mathcal{S}_0, \mathcal{S}_1, \ldots, \mathcal{S}_\alpha$, for some ordinal α such that $\mathcal{S}_\alpha$ is a final system.*

Proposition 21 shows that some final system can be derived for an atomic query Q, given a finite program P and an arbitrary but fixed computation rule R. As we will soon prove, it turns out that in our restriction of SLG resolution, for any final system $\mathcal{S}$ for Q either every subgoal in $\mathcal{S}$ is completed, or no SLG_{strat} transformation is applicable because some subgoals that are evolving in $\mathcal{S}$ contain suspended X-rules (on subgoals that are also evolving in $\mathcal{S}$). In the latter case, we say that the final system is *flummoxed*. Recall that non-succeeding subgoals cannot contain suspended X-rules in order to be completely evaluated; as a result the COMPLETION transformation cannot be applied to them. Note that a special instance of this occurs in non-ground programs where the final system may be flummoxed because some suspended X-rule of a subgoal in $\mathcal{S}$ has a selected negative literal that is not ground and is evolving.

Definition 22 Completed and Flummoxed System. Let $\mathcal{S}$ be a system. $\mathcal{S}$ is *completed* if every subgoal in $\mathcal{S}$ is completed, and is *flummoxed* if $\mathcal{S}$ is final (i.e. no transformation is applicable to $\mathcal{S}$) but not completed. □

Lemma 23. *Let P be a finite program, R be an arbitrary but fixed computation rule, and Q be an atomic query. Then for every final system $\mathcal{S}$ for Q, $\mathcal{S}$ is either completed or flummoxed.*

3.5 Subgoal Dependencies

An SLG system $\mathcal{S}$ induces a natural dependency relation between its subgoals. A subgoal A *directly depends positively* (*negatively*) on a subgoal B if B ($\neg B$) is the selected literal of an active (suspended) rule of A. A subgoal A *depends on* a subgoal B if there is a sequence of subgoals $A_0, A_1, \ldots, A_n$, such that $A_0 = A$, $A_n = B$, and A_i directly depends on A_{i+1}, for each $0 \le i \le n-1$. We say that A *depends positively on* B if A depends on B through only positive direct dependencies, and we say that A *depends negatively on* B if A depends on B and at least one of the direct dependencies is negative. Finally, we say that a subgoal is *involved in a negative loop* if it depends negatively on itself.

The dependency relation outlined above can be extended to a *Subgoal Dependency Graph* (SDG) for a system S whose vertices are non-completed subgoals in S and whose labeled links are determined by the dependency relation. Using the SDG, we can identify sets of mutually dependent subgoals as *strongly connected components* ($SCCs$). We can also speak of a *component graph* induced by a SDG as a graph whose nodes are SCCs in the SDG such that two nodes in the component graphs have a link iff there is a link between elements of the corresponding SCCs.

The following theorem states that when a final system is flummoxed, then all its subgoals that are not completed depend (directly or indirectly) on some subgoal that is involved in a negative loop.

Theorem 24. *Let P be a program, R be an arbitrary but fixed computation rule, Q be an atomic query, and $\mathcal{S}$ be a flummoxed system in an arbitrary SLG_{strat} derivation for Q. Then every subgoal in $\mathcal{S}$ that is not completed depends negatively on a subgoal in $\mathcal{S}$ that is not completely evaluated.*

3.6 Interpretation defined by an SLG system

Let P be a program, R be an arbitrary but fixed computation rule, and $\mathcal{S}$ be a system in any SLG derivation for an atomic query. As observed in [3], SLG resolution can be viewed as program transformation technique that specializes a program with respect to a query. More specifically, SLG resolution tries to transform a system of subgoals, each having an associated sequence of X-rules, into one that contains only X-rules that are answers for the subgoals. Under this prism, for each subgoal A in $\mathcal{S}$, the multiset of X-rules of A in $\mathcal{S}$ that are not disposed constitutes the set of partial answers for A. The head of each X-rule contains relevant variable bindings that have been accumulated; while the remaining literals in the rule body are yet to be solved with respect to the original program P. Each SLG system $\mathcal{S}$ represents a program $P_{\mathcal{S}}$ containing partial answers for its subgoals. It is natural, therefore, to associate with every SLG system $\mathcal{S}$ a 3-valued interpretation $I_{\mathcal{S}}$ of $P_{\mathcal{S}}$ that is constructed from $\mathcal{S}$ according to the following definition.

Definition 25. [3] Let P be a program, R be an arbitrary but fixed computation rule, and Q be an atomic query. Let $\mathcal{S}$ be a system in an SLG derivation for Q. We associate with $\mathcal{S}$ a pair $I_{\mathcal{S}} = \langle T_{\mathcal{S}}; F_{\mathcal{S}} \rangle$, where $T_{\mathcal{S}}$ and $F_{\mathcal{S}}$ are subsets of the Herbrand base H_P of P constructed as follows:

- if a (possibly non-completed) subgoal A in $\mathcal{S}$ has an answer of the form $H \leftarrow$, then every ground instance B of H is true in $I_{\mathcal{S}}$; that is, $B \in T_{\mathcal{S}}$;
- if a subgoal A in $\mathcal{S}$ is completed and B is a ground instance of A such that B is not an instance of any answer of A, then B is false in $I_{\mathcal{S}}$; that is, $B \in F_{\mathcal{S}}$;
- no other element of H_P is in $T_{\mathcal{S}} \cup F_{\mathcal{S}}$. □

In general, the interpretation $I_{\mathcal{S}}$ captures ground instances of subgoals that succeed or fail. For example, if a subgoal A succeeds in $\mathcal{S}$, then every ground instance of A is in $T_{\mathcal{S}}$, and if a subgoal A fails, then $B \in F_{\mathcal{S}}$ for every ground instance B of A. As a special case, in ground programs, a ground atom $A \in T_{\mathcal{S}}$ ($A \in F_{\mathcal{S}}$) if and only if A in $\mathcal{S}$ and A succeeds (fails) in $\mathcal{S}$. From the correctness of our restriction of SLG resolution (to be presented as Theorem 30), and the way that the interpretation $I_{\mathcal{S}}$ is constructed, it can be seen that the sets $T_{\mathcal{S}}$ and $F_{\mathcal{S}}$ are disjoint, and the following proposition is immediate:

Proposition 26. *Let P be a program, R be an arbitrary but fixed computation rule, and let $\mathcal{S}$ be a system in an arbitrary SLG_{strat} derivation for an atomic query Q. Then $I_{\mathcal{S}}$ is a consistent interpretation of P.*

Since $I_{\mathcal{S}}$ is a consistent interpretation of P, it can be viewed as valuation function of the elements of the Herbrand base H_P. In general, a ground instance of a subgoal A that is in $\mathcal{S}$ may be true, false, or undefined in $I_{\mathcal{S}}$, while all other elements of H_P that are not instances of subgoals in $\mathcal{S}$ are undefined in $I_{\mathcal{S}}$. If B is a ground atom from H_P, we denote the truth value of B in $I_{\mathcal{S}}$ as $val_{I_{\mathcal{S}}}(B)$.

We say that an interpretation $I' = \langle T'; F' \rangle$ *extends* an interpretation $I = \langle T; F \rangle$ if $T \subseteq T'$ and $F \subseteq F'$. We will denote this fact by $I \leq I'$. The following proposition states that each SLG_{strat} transformation extends the interpretation of the system to which the transformation is applied.

Proposition 27. *Let P be a program, R be an arbitrary but fixed computation rule, Q be an atomic query, and $\mathcal{S}_0, \mathcal{S}_1, \ldots, \mathcal{S}_\alpha$ be any arbitrary SLG_{strat} derivation for Q, where α is an ordinal. Then the sequence $\{I_{\mathcal{S}_i}, i > 0\}$ is monotonically increasing, i.e. $I_{\mathcal{S}_i} \leq I_{\mathcal{S}_j}$ if $i \leq j$.*

4 Correctness of SLG_{strat} for LRD-stratified Programs

Recall that for simplicity of presentation a left-to-right computation rule is used, and that the dynamic stratum of a ground atom is also defined under this left-to-right computation rule (Definition 6). All results of this section can be extended to any fixed-order computation rule.

Lemma 28. *Let P be a ground LRD-stratified program, Q be a ground atomic query, and $\mathcal{E}$ be an SLG_{strat} derivation for Q under a left-to-right computation rule. Then for any completely evaluated subgoal A in a system $\mathcal{S}$ in $\mathcal{E}$,*

$$val_{I_{\mathcal{S}}}(A) = val_{M_P}(A)$$

That is, the truth value of subgoal in $I_{\mathcal{S}}$ is the value of the subgoal in M_P.

The following lemma states that the SLG_{strat} evaluation of ground LRD-stratified programs will never result in a flummoxed system.

Lemma 29. *Let P be a ground LRD-stratified program, Q be a ground atomic query, and $\mathcal{S}_0, \mathcal{S}_1, \ldots, \mathcal{S}_\alpha$ be any SLG_{strat} derivation for Q under a left-to-right computation rule, such that $\mathcal{S}_\alpha$ is a final system. Then $\mathcal{S}_\alpha$ is completed.*

In order to prove correctness of SLG_{strat} resolution, we will want to compare the interpretation of a system $\mathcal{S}$ for a program P and query Q with the model M_P for that program. In that case, we use the notation $M_P|_{\mathcal{S}}$ to denote the restriction of M_P to (ground) atoms which unify with subgoals in $\mathcal{S}$.

Theorem 30. *Let P be a ground LRD-stratified program and Q be an atomic ground query. Then any SLG_{strat} derivation for Q under a left-to-right computation rule will produce a final system $\mathcal{S}$ such that $I_{\mathcal{S}} = M_P|_{\mathcal{S}}$.*

5 Beyond LRD-stratified programs

Recall that dynamically stratified programs differ from most stratification formalisms in that their recursive components (consisting of ground atoms) are determined dynamically in the course of a transfinite fixpoint computation. This process eliminates all irrelevant dependencies between atoms; i.e. those dependencies that can never be satisfied. As a result, dynamic stratification is very powerful: every logic program has a two-valued well-founded model if and only if it is dynamically stratified. Otherwise, in a partial well-founded model, the undefined atoms may be designated as belonging to the *ultimate dynamic stratum*.

5.1 Goal-Orientation and Relevance

Since the well-founded semantics cannot be computed using a fixed computation rule, it is non-trivial to obtain an evaluation strategy that is completely goal-directed. In particular, it is difficult given a query Q to guarantee that the computation rule will always select literals that are absolutely essential in order to prove or disprove Q. This ideal measure of relevance in computing the well-founded model is very strong, and, as it probably requires selecting literals based on knowing a priori their truth value, it is probably unobtainable for evaluation strategies whose computation rule is not guided by an oracle. Assuming that this requirement cannot be met in practice, it is then natural to ask how tight can be the relevance of an evaluation strategy for the well-founded semantics that uses a non-ideal (for example a mostly fixed-order) computation rule.

Restricting attention to the case where the evaluation encounters only ground subgoals, we describe the notion of *relevance of subgoals to a query* under a fixed computation rule (again, for simplicity of presentation we use a left-to-right computation rule). Let Q be a ground query to a program P and let P contain a rule:

$$r: \; p(\overline{t}) :- l_1(\overline{t_1}), \ldots, l_n(\overline{t_n}).$$

where $l_i(\overline{t_i}) = (\neg)s_i(\overline{t_i})$. Then, the subgoal $s_i(\overline{b_i})$ of literal $l_i(\overline{b_i})$ is relevant to Q if there is a ground instance

$$r': \; p(\overline{a}) :- l_1(\overline{b_1}), \ldots, l_i(\overline{b_i}), \ldots$$

of (the head and the first i body literals of) r such that the head $p(\overline{a})$ is relevant to Q, and all instantiated literals $l_1(\overline{b_1}), \ldots, l_{i-1}(\overline{b_{i-1}})$ are either true or undefined in the well-founded model of P. We believe that even this criterion for relevance is still very strong for all normal logic programs and appears to be equally unobtainable by non-ideal evaluation strategies. Notice, however, that all subgoals encountered during the SLG_{strat} evaluation of a query Q against a left-to-right dynamically stratified program are relevant to Q. Further evidence for this can be obtained by noticing the similarity between the notion of *relevance* sketched above and the notion of the *failing rule prefix* of Definition 1. In other words, the class of programs that are fixed-order dynamically stratified can indeed be evaluated with the tightest possible relevance to the query using a fixed computation rule. This is yet another reason for our claim that fixed-order dynamic stratification can be considered as the limit of fixed-order computation.

It is now probably easy to see that the goal-directedness and strong relevance properties of SLG_{strat} derivations make SLG_{strat} more powerful than what was claimed in the previous section. More specifically, SLG_{strat} can correctly process a query Q to a program P that is not LRD-stratified provided that subgoals belonging to the ultimate left-to-right dynamic stratum of P are not relevant to Q. This is illustrated by the following examples.

Example 3. The following program is not LRD-stratified. As discussed in Example 1 the query `p(a)` cannot be evaluated using any fixed computation rule. Consequently, its SLG_{strat} derivation using a left-to-right computation rule will result in a final system that is flummoxed.

```
p(X) :- t(X,Y,Z), ¬p(Y), ¬p(Z).
p(b).
t(a,b,a). t(a,a,b). t(b,b,a).
```

Notice however, that SLG_{strat} using a left-to-right computation rule suffices for the evaluation of query `p(b)`, since subgoal $p(a)$ is not relevant to `p(b)`. The encountered cycle through negation involving subgoal $p(b)$ is spurious and as such is eliminated by the *early completion* operation. ■

Example 4. The definition of `p/0` makes the following program non-stratified (and consequently non LRD-stratified).

```
a :- b.    b :- ¬a, p.
a.         p :- ¬p.
```

However, as subgoal p is not relevant to queries `?- a.` or `?- b.` under a left-to-right computation rule, the rule for `p/0` will not be examined, and SLG_{strat} suffices for the evaluation of these queries. ■

5.2 Towards Efficient Evaluation of the Well-Founded Semantics

By including the DELAYING, SIMPLIFICATION, and ANSWER COMPLETION transformations of SLG resolution, [2] SLG_{strat} can be naturally extended to evaluate the well-founded semantics of all normal logic programs in a goal-oriented fashion. The resulting evaluation strategy, which we call SLG_{RD} (*SLG with Reduced Delay*), is effective and inherits the properties of SLG resolution: it is sound and search space complete with respect to the well-founded partial model for all (non-floundering) queries, preserves all their three-valued stable models, and has polynomial data complexity for function-free programs.

However, through the introduction of the NEGATIVE SUSPENSION transformation, SLG_{RD} significantly restricts the space of possible SLG derivations. SLG allows an arbitrary computation rule for selecting a literal from a rule body and an arbitrary control strategy for selecting transformations to apply. This flexibility, though theoretically appealing, may result in inefficient evaluation due to unnecessary use of the DELAYING (and as a result of the SIMPLIFICATION and the ANSWER COMPLETION) transformations. Unnecessarily DELAYING ground negative literals that the computation rule stumbles on, opens up the search space by expanding subgoals that may not be relevant to the query. Contrary to SLG evaluation, SLG_{RD} suspends that computation path, and later applies the DELAYING transformation to these negative literals only if their subgoals have no left-to-right proof, i.e. only if their evaluation results in a flummoxed system. Specifically, the following theorem can be proven.

Theorem 31. *Let $\neg S$ be a selected literal in an SLG_{RD} evaluation of a ground normal logic program P under a left-to-right computation rule. Then the* DELAYING *transformation is applied to $\neg S$ if and only if S belongs to the ultimate left-to-right dynamic stratum of P.*

This theorem can be seen as addressing the question of relevance in SLG_{RD}, our restriction of SLG resolution. In principle, if a rule instance is used in a left-to-right well-founded evaluation of a query, a literal of that rule should be relevant only if it is preceded by a sequence of literals that are true or undefined in the well-founded model. As discussed in the previous section, this criterion for relevance is very strong and appears unobtainable by practical strategies that mainly use a fixed computation rule. Theorem 31 thus states our approximation to this ideal measure of relevance by using the notion of the ultimate left-to-right dynamic stratum. The discussion of the previous section hints that this approximation is tight — possibly the tightest that can be achieved using a fixed computation rule.

6 Applicability of Results

The results of Section 4 show that SLG_{strat} can form the basis of an engine to evaluate LRD-stratified programs, and the discussion of the previous section indicates that SLG_{strat} can also be used as a basis to restrict the dynamic

[2] We do not repeat their definitions here; they can be included as presented in [3].

search in engines that evaluate the well-founded semantics. In fact, SLG_{strat} has been implemented in the SLG-WAM of XSB [14]. The subset of transformations New Subgoal, Positive Return, and Completion (without the early completion of Section 3.3) suffices to evaluate definite programs. A detailed description of the their implementation, along with their integration with the SLD resolution of Prolog, is given in [17]. The addition to the SLG-WAM of negative return, negative suspension and the *early completion* operation has been described in a companion paper ([15]) and this engine forms the basis of version 1.5 of XSB. Indeed, the performance results of [15] demonstrate that LRD-stratified programs besides being evaluatable using a fixed computation rule are also amenable to an efficient implementation. In fact, the introduction of the *early completion* mechanism can *reduce* the time and space needed for evaluation of even definite programs. Based on SLG_{RD}, i.e. using the delaying transformation to break the fixed-order search only whenever the evaluation encounters a flummoxed state, we continue our effort of extending the SLG-WAM to evaluate the well-founded semantics of all normal programs.

Despite the fact that our results have been presented using (variants of) SLG resolution, we believe that the underlying ideas are potentially applicable to deductive database systems whose engines have radically different designs from XSB. For example, CORAL [11] evaluates modularly stratified programs using Ordered Search [10]. Described simply, this technique keeps a tree of dependencies and restricts magic evaluation to magic facts that are in an SCC. Both Ordered Search and its extension Well-Founded Ordered Search as originally formulated do not contain a mechanism like early completion; a mechanism that usually eliminates many spurious negative cycles, thereby reducing the need for dynamic search. It appears possible to extend these strategies to include early completion by checking whether a given magic fact is subsumed by an answer fact.

References

1. N. Bidoit and C. Froidevaux. Negation by default and unstratifiable logic programs. *Theoretical Comput. Sci.*, 78(1):85–112, January 1991.
2. R. Bol and L. Degerstedt. Tabulated resolution for well founded semantics. In D. Miller, editor, *Proceedings of the 1993 International Symposium on Logic Programming*, pages 199–219, Vancouver, October 1993. The MIT Press.
3. W. Chen and D. S. Warren. Tabled Evaluation with Delaying for General Logic Programs. *J. ACM*, 43(1):20–74, January 1996.
4. J. Dix. Classifying semantics of logic programs. In A. Nerode, W. Marek, and V. S. Subrahmanian, editors, *Logic Programming and Mon-Monotonic Reasoning, Proceedings of the first International Workshop*, pages 166–180, Washington D.C., July 1991. The MIT Press.
5. M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In R. A. Kowalski and K. A. Bowen, editors, *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pages 1081–1086, Seattle, August 1988. The MIT Press.
6. S. Morishita. An Extension of Van Gelder's Alternating Fixpoint to Magic Programs. *J. Comput. Syst. Sci.*, 52(3):506–521, June 1996.

7. H. Przymusinska and T. C. Przymusinski. Weakly Stratified Logic Programs. *Fundamenta Informaticae*, 13(1):51–65, March 1990.
8. T. C. Przymusinski. Every Logic Program has a Natural Stratification and an Iterated Least Fixed Point Model. In *Proceedings of the Eighth ACM Symposium on Principles of Database Systems*, pages 11–21, Philadelphia, Pennsylvania, March 1989. ACM.
9. T. C. Przymusinski. Well-Founded and Stationary Models of Logic Programs. *Annals of Mathematics and Artificial Intelligence*, 12:141–187, 1994.
10. R. Ramakrishnan, D. Srivastava, and S. Sudarshan. Controlling the search in bottom-up evaluation. In K. Apt, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 273–287, Washington D.C., October 1992. The MIT Press.
11. R. Ramakrishnan, D. Srivastava, and S. Sudarshan. CORAL — Control, Relations, and Logic. In *Proceedings of the 18th Conference on Very Large Data Bases*, pages 238–249, Vancouver, Canada, August 1992.
12. K. A. Ross. A procedural semantics for well-founded negation in logic programs. *Journal of Logic Programming*, 13(1):1–22, May 1992.
13. K. A. Ross. Modular Stratification and Magic Sets for Datalog programs with Negation. *J. ACM*, 41(6):1216–1266, November 1994.
14. K. Sagonas, T. Swift, and D. S. Warren. XSB as an Efficient Deductive Database Engine. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, pages 442–453, Minneapolis, Minnesota, May 1994.
15. K. Sagonas, T. Swift, and D. S. Warren. An Abstract Machine for Fixed-Order Dynamically Stratified Programs. In *Proceedings of the Thirteenth International Conference on Automated Deduction*, LNAI, New Brunswick, New Jersey, July 1996. Springer-Verlag. To appear.
16. P. J. Stuckey and S. Sudarshan. Well-Founded Ordered Search. In R. Shyamasundar, editor, *Proceedings of the 13th Conference on Foundations of Software Technology and Theoretical Computer Science*, number 761 in LNCS, pages 161–171, Bombay, India, December 1993. Springer-Verlag.
17. T. Swift and D. S. Warren. An abstract machine for SLG resolution: Definite Programs. In M. Bruynooghe, editor, *Proceedings of the 1994 International Symposium on Logic Programming*, pages 633–652, Ithaca, November 1994. The MIT Press.
18. A. Van Gelder. The Alternating Fixpoint of Logic Programs with Negation. In *Proceedings of the Eighth ACM Symposium on Principles of Database Systems*, pages 1–10, Philadelphia, Pennsylvania, March 1989. ACM.
19. A. Van Gelder, K. A. Ross, and J. S. Schlipf. The Well-Founded Semantics for General Logic Programs. *J. ACM*, 38(3):620–650, July 1991.