The SLG-WAM: A Search-Efficient Engine for Well-Founded Evaluation of Normal Logic Programs

A DISSERTATION PRESENTED
BY
KONSTANTINOS SAGONAS

TO
THE GRADUATE SCHOOL
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
IN
COMPUTER SCIENCE
STATE UNIVERSITY OF NEW YORK
AT STONY BROOK

August 1996

Abstract of the Dissertation

The SLG-WAM: A Search-Efficient Engine for

Well-Founded Evaluation of Normal Logic Programs

by

Konstantinos Sagonas

Doctor of Philosophy

in

Computer Science

State University of New York at Stony Brook

1996

We investigate ways of bringing systems based on ideas from logic programming closer to the ideals of the paradigm. We conservatively extend Prolog implementations to overcome their susceptibility to infinite loops, and their unacceptable performance caused by repeated subcomputations. Moreover, we improve the expressive power of systems whose query language is based on logical rules with negation by supporting evaluation of queries according to the well-founded semantics. We focus on how to achieve these ambitious goals without sacrificing the search-efficiency of the evaluation strategy, or the performance of the underlying abstract machine.

Higher-level issues related to the search-efficiency of evaluation strategies for the well-founded semantics are addressed by proving a bound on the non-determinism required by the computation rule. Lower-level issues that are sometimes overlooked, albeit necessary for the efficient evaluation of practical programs (and in many cases today for relevance of research in computer science), are addressed by developing the SLG-WAM, an abstract machine for the efficient evaluation of normal programs according to the well-founded semantics. The SLG-WAM conservatively extends the standard basis of Prolog implementations to include features that are necessary to support the enhanced functionality.

We believe that the SLG-WAM can serve as a conceptual framework for future implementation efforts aiming to incorporate into one system ideas from logic programming, deductive databases, and non-monotonic reasoning.

The work was done in the framework of the XSB system, various aspects of which are also briefly described.

To all those that wanted to continue their studies
but were prevented by financial difficulties,
and to my parents that were among them.

Contents

List of Figures

List of Tables

Acknowledgments

First and foremost, my thanks go to my advisor, David S. Warren. His work on logic programming was the reason I chose to come to Stony Brook; meeting him was the reason I decided to stay here; the experience of working with him during these last five years is the reason I know I will never regret my decision. I will also never forget his constant excitement about research, his amazing talent of explaining issues simply, his deep insight into problems. Besides his constant support, I benefited enormously from the numerous stimulating discussions and enlightening conversations we had. To put it in a nutshell, he is the ideal advisor. I sincerely hope I can someday repay all the enjoyment I got out of our interaction by following his example when I become a professor myself.

David, words are not enough... what am I going to do without our weakly research meetings?

For almost four and a half years, Terry Swift has been a very good friend, a pioneer XSB developer, an excellent co-author of many papers, a source of support, and in some occasions an unofficial advisor. Among other things, he is the one to blame for suggesting this dissertation topic to me and convincing me to work on it. I owe him and sincerely thank him for this.

Coming to a leading research institution in the areas of logic programming, deductive databases and automated deduction, I was fortunate and honoured to have in my dissertation committee world class researchers of these fields. I have personally benefited from I. V. Ramakrishnan's involvement in my research and in the XSB group. Besides his tireless effort to obtain funding for the group, he has been a constant source of support and ideas for extending our research. Sincere thanks are also due to Michael Kifer, Leo Bachmair and Ken Ross for their comments, questions and encouragement.

Stimulation from fellow students makes up a significant part of one's graduate experience. I was fortunate enough to spend long hours working closely together with the following talented group of people: Steve Dawson, Juliana Freire, C. R. Ramakrishnan and Prasad Rao, all members of the XSB group, that I sincerely thank for the exciting research environment that they established around the lab.

Thanks are also due to Brian Tria for letting me take matters into my own hands when something went wrong in the lab, to Kathy Germana and Betty Knittweis for their help and care. I will always be endebted to the late Peggy Thomas without whose commitment to helping students handle administrative matters I would not have come to Stony Brook.

Besides my interaction with members of the XSB group, my Ph.D. years would not be resembling a life had they been without the warmth and company of my dearest friends: Oleg, Michael, Ulf, Dimitris, George, Vaggelis, and the affection of the cats in my house: Meli and Viny.

More importantly, however, it would not have been the same if my friend and lover Sophia had not been as understanding as possible when I told her that I would need to spend four to five years away from her for my Ph.D. I would like to thank her for always patiently listening to my excitement, fatigue, complaints and frustrations. For still putting up with my absenses. For just being there... especially since that usually was the other end of the phone line. It has been very hard to stay these long years away from you, but you were my source of inspiration as well as a very compelling motivation to finish as soon as possible.

Finally, I would like to extend my thanks to Richard Larson. I enjoyed working with him on *Syntactica* and *Semantica* projects which during the first four years of my stay provided me with the necessary for a graduate student financial support. On a more personal level, I will never forget many of our discussions on computational linguistics, the nature of science and philosophy in general. Among other things, long before I started writing my Ph.D. dissertation he was the one that pointed out to me that "you do not ever *finish* writing your thesis, there comes a time that you simply have to *stop* writing it". He was right!

Κωστής Σαγώνας

This research was supported in part by a Catacosinos Fellowship for Outstanding Contributions to Computer Sciences.

Preface

This dissertation is the compilation and adaptation of three papers [62, 60, 59] which have previously appeared or are about to appear in international conference proceedings. In compiling them we have tried to make the notation and terminology as coherent as possible. Occasionally, some material was intentionally duplicated across sections to make the individual parts more self-contained. Some familiarity with the architecture of the WAM [80] (its data areas in particular) is however expected throughout Chapters 4–6.

The design and implementation of the SLG-WAM is based on the abstract machine for the tabled evaluation of definite programs, which was initially described by Swift and Warren in [69] at ILPS-94 (this paper has since been superseded by [61]). Since this dissertation aims at being the first complete description of the SLG-WAM, we have included in Section 4.5.3 some necessary material from [50] which represents joint work. The SLG-WAM forms the basis of the freely available XSB system [58] which, in addition to having been installed in several hundeds of sites around the world, is being widely used in research [19] and has been used to implement several large systems [35, 36].

Chapter 1

Introduction

1.1 Statement of the Problem

During the past fifteen years, much effort has been aimed at increasing the expressive power of logic programming and deductive databases. One of the most investigated directions has been the introduction of *negation* in the "conditional" part of the rules defining intentional predicates. This introduction of negation while being a very natural and simple idea happens to raise major problems as far as the definition of the declarative semantics of logic programs is concerned. Consequently, extensive research has been conducted on this issue [26, 34, 73, 5, 75, 46, 22, 54, 49]. During pretty much the same period as this line of research was conducted, the connections between logic programming semantics and logics for non-monotonic reasoning such as *default logic*, *autoepistemic logic* and the various versions of *circumscription* have attracted equally strong interests [26, 28, 37, 29, 39, 48].

Despite all these steady advancements concerning the semantical underpinnings of logic-based languages and all these years of fruitful research in increasing the power of evaluation strategies for these semantics, there has not been an analogous amount of effort aiming at developing frameworks upon which efficient implementations of these semantics can be based. As a result, it is perhaps not very surprising that the most popular resolution method to date is still SLDNF and that Prolog remains the most popular implementation of a language based on logic. As proven by its widespread acceptance, the Prolog/SLDNF paradigm is extremely powerful for *programming*. However this paradigm suffers from drawbacks which have prevented its extension into other areas such as deductive databases or non-monotonic reasoning that require a higher level of declarativity than Prolog can offer. A strong claim can be made that these drawbacks are rooted at the severe practical and semantic limitations of this paradigm. For instance, Datalog programs do not necessarily terminate under SLD. Furthermore, SLD may repeatedly evaluate the same literal in a rule body leading to unacceptable performance. Bottom-up evaluation methods can address these limitations, and a variety of magic sets/templates [4, 51] and tabling methods [71, 77] have been devised to overcome these problems associated with definite programs. However, another equally important limitation of the Prolog/SLDNF paradigm remains: when negation is introduced, the meaning of normal logic programs as represented by the *Clark's Completion* [16] semantics for SLDNF is far from satisfactory.

To overcome this semantic drawback, various approaches for assigning meaning to logic programs with negation have been proposed in the literature. The *well-founded semantics* (WFS) [75] is the leading candidate among semantics based on a three-valued logic and since its introduction has gained wide acceptance among the logic programming and deductive database communities. This is partly due to the fact that the WFS ensures a proper behavior under both positive and negative recursion, and can be characterized as the only semantics for normal logic programs which is both *rational* and *cumulative* [22]. Furthermore, the WFS naturally extends the iterated model semantics for stratified programs to arbitrary normal programs in such a way that it always gives them a *unique, skeptical* minimal model. That is, the well-founded model posits as *unknown* atoms which are deemed true or false in other formalisms such as stable models [29]. Since most other semantics agree with what is true or false under the WFS, this skepticism makes the well-founded model not only useful in itself, but also suitable as a basis for other forms of non-monotonic reasoning. For instance, since answers of ground queries to Datalog programs with negation can be computed in time that is polynomial in the size of the database [75], and given that the problem of even determining whether stable-models exist is intractable [39], the well-founded model can be used to restrict the search space of stable-model computation to those atoms whose truth value is unknown under WFS.

Despite these properties, the well-founded semantics is non-trivial to compute efficiently. This observation is based on fundamental aspects of computing normal logic programs. The first difficulty arises because an evaluation of a negative query has to wait until that query is completely evaluated before succeeding — conceptually the fixpoint check mechanism of magic templates or tabling methods must be extended from a check of the state of computation in the definite case, into a check of dependencies between subgoals. The second, more fundamental, difficulty arises from the inability of a fixed computation rule to evaluate the WFS of all normal programs (or even of many classes of stratified programs [56]). As a result, early evaluation techniques for the WFS such as SLS-resolution [45, 55] either employed a computation rule that was ideally guided by an oracle (and hence were only theoretical constructs), or were not goal-directed (such as the alternating fixpoint techniques of [74, 40]).

Maximal adherence to a fixed computation rule is a prerequisite for the efficient evaluation of queries under the WFS. First, evaluation strategies based on 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, a claim can be made that even the notion of goal-directedness is probably ill-defined in the absence of a fixed computation rule.

1.2 Overview of the Solution

Taking the above difficulties at face value may lead to a conclusion that the well-founded semantics cannot be efficiently computed — in other words, that normal programs, unrestricted by stratification constraints, are not suitable for general programming but only for specification or knowledge representation. We hold that despite the difficulties there is no intrinsic reason that this should be the case. We substantiate our claim in two steps.

We first address issues related to the *relevance, search-efficiency*, and *goal-directedness* of evaluation strategies for the well-founded semantics. We do so by defining a new stratification class, *left-to-right dynamically stratified* (*LRD-stratified*) *programs*, a powerful subset of normal logic programs that properly includes *left-to-right weakly stratified programs* [44], effectively including all *modularly stratified* [56] programs as well. We also introduce yet another evaluation method, SLG_{RD} (SLG with Reduced DELAYING), a variant of SLG [14] which is based on tabling. As such, SLG_{RD} inherits many of SLG's properties. In particular it is sound and complete for the well-founded semantics, terminates for all normal logic programs with the *bounded term-size property* (as defined by Van Gelder in [73]), and evaluates ground queries to Datalog programs with polynomial data complexity. In addition, SLG_{RD} significantly restricts the space of possible SLG derivations by tightening the relevance of subgoals that are resolved. This strong relevance property of SLG_{RD} permits us to show a bound on the non-determinism of the computation rule required for the evaluation of normal programs (Theorem 3.5.1). More specifically, we prove that the fixed left-to-right computation rule need not be broken for the evaluation of LRD-stratified programs. This is the subject of Chapter 3.

The second (and bigger) part of this dissertation focuses on lower-level issues that are necessary for the efficient evaluation of queries under the well-founded semantics. We *conservatively extend* the standard basis of Prolog implementations, the Warren Abstract Machine (WAM) [80], to develop an abstract machine framework for the efficient implementation of SLG_{RD} (and variants of SLG resolution in general). By basing our abstract machine for well-founded negation on the WAM, we are able to combine the flexibility and well-developed programming environment of standard Prolog with the declarative advantages of SLG_{RD}. The result is the ability of the programmer (or optimizer) to choose a mixture of SLDNF and SLG_{RD} execution for predicates, depending on which is most appropriate. The design of the SLG-WAM is developed, and sometimes reconstructed, in three consecutive chapters, (4, 5, and 6), in such a way that the presentation of the algorithms and data structures of the abstract machine is done for each conceptually separate part of the engine design: the classes of definite, LRD-stratified, and normal programs, respectively.

Furthermore, in Chapter 7 we provide experimental evidence that *programming* is feasible over the class of all normal logic programs. More specifically, we show that the features needed for the evaluation of normal logic programs under the WFS slow Prolog execution only minimally, and that for Datalog programs with negation (that terminate under Prolog's evaluation strategy), tabled execution in the SLG-WAM is competitive in speed with

Prolog. To put the SLG-WAM's performance of well-founded negation in perspective, we finally compare this overhead to that of another implemented evaluation strategy, Ordered Search [52] and attempt an analysis of the differences between tabled-based and magic-style implementations of negation.

Chapter 2

Preliminaries

To make this dissertation as self-contained as possible, after a short section on notations and terminology that we employ, we briefly overview basic results on the semantics of negation in logic-based languages. More detailed discussions concerning the approaches to negation in logic programming, deductive databases, and non-monotonic reasoning can be found in the survey papers [65, 5, 23, 2], upon which we base parts of our overview.

2.1 Notations and Terminology

We assume the standard terminology of logic programming from [38]. The following definitions extend this terminology to three-valued models. The *Herbrand base* H_P of a program P is the set of all ground atoms with predicates occurring in P that can be constructed from the constants and function symbols appearing in P. 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. Let $\mathtt{f}, \mathtt{u}, \mathtt{t}$ be the truth values "false", "undefined", and "true" ordered by the *truth ordering* $\mathtt{f} < \mathtt{u} < \mathtt{t}$. 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 $\{\mathtt{f}, \mathtt{u}, \mathtt{t}\}$ of truth values. We will also use the concept of a *well-supported interpretation* I. Intuitively, I is well-supported if every ground atom $A \in I$ has an explanation which does not use A (see [2] for a formal definition). For example, for the program $P_1 = \{\mathtt{p} :- \neg\mathtt{q}; \mathtt{q} :- \mathtt{q}\}$, the model $\{\mathtt{p}, \neg\mathtt{q}\}$ is well-supported, whereas $\{\neg\mathtt{p}, \mathtt{q}\}$ is not.

2.2 Semantics of Negation in Logic-Based Languages

Simple forms of logic, such as Horn clause rules (rules without negative literals in the body), have a natural meaning which is the least fixpoint, or provenly unique minimal model of the program [72]. However, this form of rules is often too limited since many interesting queries require use of more general forms of rules, such as those in which one or more subgoals appear negatively. This is true regardless of whether one is interested in logic programming, or in deductive database languages. Adherence to classical deductive logic rarely offers the intuitively correct meaning of the rules. The problem is that as soon as negative subgoals are introduced, there is rarely a unique minimal model; in fact, it is normal to have more than one (cf. program P_1 on page 5). In conventional logic, the rules are assumed to "mean" whatever can logically be inferred from these rules and nothing more. This is the same as saying that the "meaning" of a program is the intersection of all its minimal models. However, this intersection is rarely what the programmer intuitively expects these rules to "mean". Thus, research in logic-based languages with negation can be seen as a series of proposals to define one particular *minimal model* as the meaning of logical rules. As a result, a variety of approaches to defining the "right" meaning of such rules have been explored, and there are important classes of programs for which there is a widely accepted meaning. Principal approaches among these are the various classes of *stratified negation* that we briefly review below (and we extend in the following chapter), negation based on *well-founded partial models* [75], and on *stable-model semantics* [29]. In other cases, the proper choice of meaning is less clear and, in any case, beyond the scope of this document.

Before presenting the approaches to negation mentioned above, we note that the subject of negation in logic-based languages is closely related to *non-monotonic reasoning* as discussed in the literature of Artificial Intelligence. There is a general tendency in the logic programming and deductive database communities to value more the issues of efficiency, i.e., how fast queries can be answered using a particular definition of the "correct" minimal model. The AI approaches, on the other hand, tend to focus on the correctness of the model, regardless of its tractability. This difference can be seen from a comparison of the well-founded and the stable-model approaches of assigning meaning to logic programs (cf. Section 2.2.2). Another, perhaps more subtle, difference is that in logic programming languages and deductive databases it is natural to specify a *single* canonical model as the semantics of the program; query evaluation then becomes the process of finding an answer to the query with respect to that canonical model. On the other hand, in artificial intelligence applications the same query may have different *sets of answers* in different possible scenarios, worlds, or in "sets of beliefs that a rational agent might hold" [29] (i.e., in different stable models). The multiplicity of stable models lends itself naturally to several forms of non-monotonic reasoning, such as abduction and skeptical reasoning.

2.2.1 Semantics based on Stratified Negation

The least controversial of the approaches to assigning meaning to logic programs with negation are in cases where the use of negation is *stratified*, i.e., where there is no recursion involving negated subgoals. The basic idea, arrived at independently by Apt, Blair, and Walker [1], and Van Gelder [73], is that when a logic program is stratified, we can find an order for the predicates, so we may solve for a predicate p only after we have solved for all predicates on which p depends negatively. That is, a program is *stratified* if there is an assignment of ordinal levels to *predicates* such that whenever a predicate appears negatively in the body of a rule, the predicate in the head of that rule is of strictly higher level, and whenever a predicate appears positively in the body of a rule, the predicate in the head has at least that level. Stratified programs have a unique "natural" Herbrand model M_P, which arguably should be used to represent the intended meaning of such programs. The result of computing the predicates in this way is often called the *perfect model* [46].

Przymusinski, in [46], extended the class of stratified logic programs to the wider class of *locally stratified* programs, where predicates are allowed to depend negatively on themselves as long as no negative cycles between ground atoms occur in the Herbrand instantiation of the program. Unfortunately, local stratification is not a decidable property of a program. Surely however, every stratified logic program is locally stratified regardless of the set of its constants. Every locally stratified program has exactly one perfect Herbrand model, which — for stratified programs — coincides with the "natural" model M_P. The class of perfect models of a logic program has many natural and desirable properties. Among the most important of these is that all perfect models are minimal, and for definite programs the notions of a minimal and a perfect model coincide. As a result, the *perfect model semantics* has been generally adopted as the basis for several experimental deductive database systems.

Example 2.2.1 Consider the program P consisting of the rule

```
win(X) :- move(X,Y), ¬win(Y).
```

together with some facts about `move/2`. P is a stalemate game program in which a position `X` is considered a winning position if there is a move from `X` to a position `Y` which is not a winning (i.e., is a losing) position.

P is not stratified or even locally stratified. In fact, Kolaitis in [33] has shown that no stratified function-free program can express the intended semantics of P; in other words unstratified programs, as a class, have strictly greater expressive power than stratified. The intuition behind this expressiveness result is that stratified programs have a fixed number of strata through which one can recurse through negation. This lack of expressive power was the main motivation for further refining the class of stratified programs. ■

Ross introduced *modular stratification* [56] in an attempt to identify a subset of logic programs with unambiguous semantics which could be evaluated efficiently. Here, efficiency mainly concerned the *"magic-sets" transformation* [4, 51] that adds goal-orientation to pure bottom-up evaluation strategies (i.e., it allows queries to be answered without looking at parts of the database that are irrelevant to the query). Informally, for a logic program to be *modularly stratified*, it should be possible to divide its *predicates* into *modules* with the following properties:

1. Modules should be orderable in such a way so that predicates in a module depend only on predicates in that module and on previous ones. In other words, there may be recursion among predicates, but it should take place solely within modules.

2. Each module should have a locally stratified perfect model when its rules are instantiated, and all subgoals whose predicates are in previous modules are treated as true or false according to the well-founded model for each module.

Breaking a program up into its components provides a modular framework for defining program semantics; predicates appearing in rule heads in a given component should be well defined for all values of lower-level predicates. Thus, we can compute the locally stratified model for each module, in order, using the results from previous modules to determine the truth or falsehood of subgoals that are outside the module. Since each module is thus given a two valued model, there is no ambiguity as to the truth value of subgoals outside the module. For example, it is shown in [56] that program P of example 2.2.1 is modularly stratified when the `move/2` relation is acyclic, that is, when the game cannot have repeated positions.

The class of logic programs with well-defined, unambiguous semantics is fairly broad and is not limited to the class of stratified, locally stratified, or even modularly stratified programs. As a natural extension of the stratification classes, Przymusinska and Przymusinski, in [44], introduced the wider class of *weakly stratified* programs, and an extension of the class of perfect models to the class of *weakly perfect models*. All weakly stratified programs have weakly perfect models, but the class of programs with weakly perfect models is more general that the class of weakly stratified programs. The main idea behind this extension is to remove irrelevant relations from the dependency graph of a logic program and to substitute *components* of the dependency graph for its vertices in the definition of stratification and perfect models. More precisely, *weakly stratified* programs allow a predicate to depend negatively on itself as long as no *ground atom* depends negatively on itself once all instantiated rules that have subgoals that are known to be false or heads that are known to be true are removed.

We note that weakly stratified logic programs cannot, in general, be evaluated using a fixed literal selection strategy. For example, when evaluated with a left-to-right rule, a subgoal may encounter a loop through negation even though some later subgoal fails. Since subgoal-at-a-time computation is important for efficient evaluation, it is desirable to identify a subclass of stratified programs that can be evaluated one subgoal at a time. This issue is examined in the following chapter where we further extend the stratification classes.

2.2.2 Semantics based on Well-Founded and Stable Models

There are many interesting logic programs that are not stratifiable, but for which a natural, unambiguous semantics exists. The *well-founded semantics* and the *stable model semantics* are the two leading, and closely related, proposals for defining the semantics of normal logic programs, whether stratified or not. In fact, for (weakly) stratified programs, they both coincide with the (weakly) perfect model semantics.

Well-Founded Semantics Based on the notion of the *greatest unfounded set*, Van Gelder, Ross and Schlipf in [75] presented a non-constructive definition of a three-valued (partial) model of P, called the *well-founded (partial) model of* P, which we briefly review below. For somewhat more constructive definitions of the well-founded semantics refer to [74, 45, 6].

Unfounded sets provide the basis for negative conclusions in the well-founded semantics.

Definition 2.2.1 (*Unfounded Set*) Let P be a program, H_P its associated Herbrand base, and let I be a consistent set of ground literals whose atoms are in H_P. We say that $U \subseteq H_P$ is an *unfounded set of* P *with respect to* I if each atom $p \in U$ satisfies the following condition: For each instantiated rule r of P whose head is p, at least one of the following holds:

1. The complement of some literal in the body of r is in I.
2. Some positive literal in the body of r is in U.

The union of all sets that are unfounded with respect to I, is also an unfounded set, and is called the *greatest unfounded set of* P *with respect to* I. □

Definition 2.2.2 Transformations T_P, U_P, and W_P from sets of literals to sets of literals are defined as follows:

T_P is the usual *immediate consequence* operator; $p \in T_P(I)$ if and only if there is some instantiated rule r of P such that r has head p and each literal in the body of r is in I.

$U_P(I)$ is the greatest unfounded set of P with respect to I.

$W_P(I)$ is defined as $T_P(I) \cup \neg \cdot U_P(I)$. □

It is immediate to show that T_P, U_P, and W_P are monotonic transformations, and therefore have a least fixpoint. The well-founded (partial) model of P is the least fixpoint of W_P. Note that the well-founded model is, in general, three-valued: a ground atom may appear positively (i.e., be *true*), negatively (i.e., be *false*), or not at all (i.e., be *undefined*) in the well-founded model.

Since its introduction, the well-founded semantics is the leading candidate among the three-valued semantics that have been proposed in the literature and has been generally viewed as the desired choice of semantics of normal logic programs from a deductive database point of view. This partly due to the fact that the well-founded semantics extends the iterated model semantics for stratified programs to arbitrary normal programs in such a way that it gives them a unique minimal model.

Stable Model Semantics Gelfond and Lifschitz in [29] proposed an elegant definition of a *stable model* of a logic program as a model that is able to reproduce itself with a certain natural *stability transformation* (see [29, 2] for details). Stable models are two-valued. A given program may have zero, one or many stable models. This multiplicity of stable models corresponds naturally with the intuition of *rational beliefs* from autoepistemic logic, reflects the *uncertainty* of conclusions that could be drawn from a program, and introduces an extra level of *non-determinism* which may be very desirable in certain applications. In some cases, however, a "local" uncertainty can destroy too much information: for example, if P is a stratified program in which the relation symbol p does not occur, then $P \cup \{\texttt{p} :- \neg\texttt{p}\}$ has no stable model. Thus, the information contained in P is not reflected in the stable model semantics, although it is not related to the uncertainty about the truth value of p. Three-valued extensions of stable models, *partial stable models*, overcome this problem [30, 47, 49].

When a program has a *unique* stable model, it is the natural one to associate with the program. In [75] it is shown that a well-founded total model is always the unique stable model, but not *vise versa*; i.e., it is possible that a program has a unique stable model, yet its well-founded model may not be total. Furthermore, Fages in [25] shows that concepts of *well-supported* (cf. page 5) and stable models coincide.

So, which semantics should I use? The subject of what is the "right" semantics of logic programs with negation has been extensively studied, discussed, and debated. However, as the subject has received more attention, anomalies have surfaced for both the well-founded and stable model approaches. Consequently, there have been a number of proposals modifying or relating the well-founded and stable model semantics. For example, Saccà and Zaniolo [57] look at intersections of stable models. Przymusinski [47] presents a three-valued extension to the original two-valued definition of stable models, and shows that this three-valued extension coincides with the well-founded model of the program. Finally, Baral and Subrahmanian consider *sets* of stable models as a meaning for programs [3].

Despite similarities between semantics of logic programs identified in the literature, an important remaining difference between the well-founded and stable model approaches is *tractability*. For Datalog programs, well-founded models can be computed in time that is polynomial in the size of the database. However, unless the well-founded model is total (so that it coincides with the unique stable model of the program), there is no known polynomial time algorithm for finding a stable model or for even telling if one exists. Furthermore, it is unlikely that such an algorithm will ever exist as the problem of determining whether a propositional program has a stable model has been shown to be NP-complete [39].

Chapter 3

Search-Efficient Tabled Evaluation of the Well-Founded Semantics

3.1 On Evaluating Normal Logic Programs using a Fixed Computation Rule

It is known that the well-founded semantics (WFS) combines the features of both the *stable model* [29] and *partial Clark completion* [49] 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* [22]. 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 [45, 55]. Unfortunately, SLS-resolution is not effective in general. As formulated by Przymusinski [45] SLS-resolution uses an oracle to select literals from lower dynamic strata, while Global SLS-resolution as formulated by Ross [55] requires a positivistic computation rule and that all negative literals are ideally evaluated in parallel. These departures from fixed computation rules are naturally reflected in more practical strategies for evaluating the WFS of normal logic programs [8, 14, 40, 66]. 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 [14], 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 [66], magic templates with Ordered Search [52] are used for the first phase, while an alternating fixpoint similar to that of [74] is used for the second.

The limitations of fixed-order computation[1] can be seen clearly in the following example.

Example 3.1.1 ([56]) Consider the following program and query `p(a)`.

```
p(X) :- t(X,Y,Z), ¬p(Y), ¬p(Z).
p(b).
t(a,b,a).
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 fixed computation rules, it is natural to try to determine what their 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 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. [53]). 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 3.1.2 The following program is neither *modularly stratified*, nor *weakly stratified*.

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

[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 this document. All results can be easily extended to any fixed computation rule.

Notice, however, that it has a two-valued Well-Founded Model, $T = \{s\}$, $F = \{p, q, r\}$, that can be computed using a fixed computation rule, as we will show later in this chapter. ■

To our knowledge, the only stratification class that includes the program in Example 3.1.2 is *dynamic stratification* of [45] (also independently introduced by Bidoit and Froidevaux as *effective stratification* in [6]). From the perspective of the well-founded semantics, dynamic stratification is extremely powerful since all normal logic programs with total (i.e. 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 standard stratifications 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, our objective 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 chapter 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.

3. Based on SLG_{Strat}, we introduce SLG_{RD}, a variant of SLG resolution that evaluates normal logic programs according to the well-founded semantics. SLG_{RD} inherits most of the properties of SLG while at the same time restricts, whenever it is possible to do so, the space of possible SLG derivations to those that use a deterministic fixed computation rule; consequently minimizing the need for dynamic search.

The structure of the rest of this chapter is as follows. We define LRD-stratified programs in Section 3.2. We then introduce SLG_{Strat} in Section 3.3, and in Section 3.4 show the completeness of SLG_{Strat} for ground LRD-stratified programs. Section 3.5 addresses the issue of relevance and goal-orientation of evaluation strategies, introduces SLG_{RD}, and argues that SLG_{RD} constrains the search space of possible derivations of SLG resolution. We conclude in Section 3.6 with a discussion of the use and the potential applicability of our results to other evaluation methods for stratified negation. Proofs for theorems in this chapter appear in Appendix A.

3.2 Left-to-Right Dynamically Stratified Programs

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

We first adapt Przymusinski's definition of the iterated fixed point from [45] 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 3.2.1). 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 introduction of 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 3.2.1 *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 3.2.2 [45] *Let* $I = \langle T; F\rangle$ *be an interpretation, we define*

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

Proposition 3.2.1 [45] *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 3.2.3 [45] *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 3.2.4 (*Iterated Fixed Point*) [45] *Let:*

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

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 since it may overestimate the set of undefined atoms. 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.

3.2.2 Left-to-Right Dynamic Stratification

As observed in [45], the iterated fixed point induces a natural stratification on a program P. Similarly, the iterated fixed point under a fixed computation rule induces an equally natural stratification.

Definition 3.2.5 (*Left-to-Right Dynamic Stratum*) *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 3.2.6 (*LRD-Stratified Program*) *A program is* left-to-right dynamically stratified *if its ultimate stratum S_δ is empty.* □

Theorem 3.2.2 *Let P be a left-to-right dynamically stratified program. Then the interpretation $J_{IF(P)}$ is a minimal 2-valued model of P.*

Proof: That $J_{IF(P)}$ is two-valued follows immediately from definitions 3.2.5 and 3.2.6. The proof that $J_{IF(P)}$ is a minimal model is a simple restriction of a similar proof in [45]. ◇

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. For LRD-stratified programs, it also coincides with the well-founded [75] and the unique stable model [29] of the program P.

Example 3.2.1 The iterated fixed point of the program of Example 3.1.2 is as follows:

$$\begin{aligned} J_0 &= \langle \emptyset; \emptyset \rangle; \\ J_1 &= \langle \emptyset; \{\mathtt{p}, \mathtt{q}, \mathtt{r}\} \rangle; \\ J_2 &= \langle \{\mathtt{s}\}; \{\mathtt{p}, \mathtt{q}, \mathtt{r}\} \rangle; \end{aligned}$$

Since both sets T_{J_2} and F_{J_2} are empty, $J_2 = J_{IF(P)}$. This program is LRD-stratified, and J_2 is its well-founded model M_P. Its dynamic strata are $S_0 = \{\mathtt{p}, \mathtt{q}, \mathtt{r}\}$ and $S_1 = \{\mathtt{s}\}$. ■

Example 3.2.2 Suppose that P is the program of Figure 1(a).

The iterated fixed point of P is shown in Figure 1(b). Its *ultimate left-to-right dynamic stratum* $S_3 = \{\mathtt{p}, \mathtt{q}\}$ is not empty; consequently the program is not LRD-stratified. Intuitively, the program is not LRD-stratified since the rules defining `p/0` and `q/0` neither

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

(a) Program

$$\begin{aligned} J_0 &= \langle \emptyset; \emptyset \rangle; \\ J_1 &= \langle \emptyset; \{\mathtt{a}\} \rangle; \\ J_2 &= \langle \{\mathtt{b}\}; \{\mathtt{a}\} \rangle; \\ J_3 &= \langle \{\mathtt{b}\}; \{\mathtt{a}, \mathtt{c}\} \rangle; \end{aligned}$$

(b) Iterated fixed point

Figure 1: *A dynamically but not left-to-right dynamically stratified program*

succeed nor contain a failing prefix. Note, however, that the program has a well-founded model $M_P = \langle \{\mathtt{b}, \mathtt{p}\}; \{\mathtt{a}, \mathtt{c}, \mathtt{q}\} \rangle$ which is total and therefore P is dynamically stratified. If the rule for `q/0` were substituted by the rule `q :- a, ¬p.` then the program would be LRD-stratified. ■

Although many programs (as the one of Example 3.2.2) are compile-time reorderable so that they become left-to-right dynamically stratified, there are programs for which any consistent sequential reordering of their literals (including a left-to-right ordering) is impossible. Example 3.1.1 is an instance of such a program.

3.2.3 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* [44], or in *modular stratification* [56]. 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 Figure 2 where arrows indicate class inclusion.

Most of the relationships between the stratification classes presented in Figure 2 are well-known in the literature. Theorem 3.2.3 substantiates the placement of LRD-stratification within the stratification hierarchy. The proof of Theorem 3.2.3, along with supporting definitions of the class of *left-to-right weakly stratified programs*, is given in Appendix A.2.

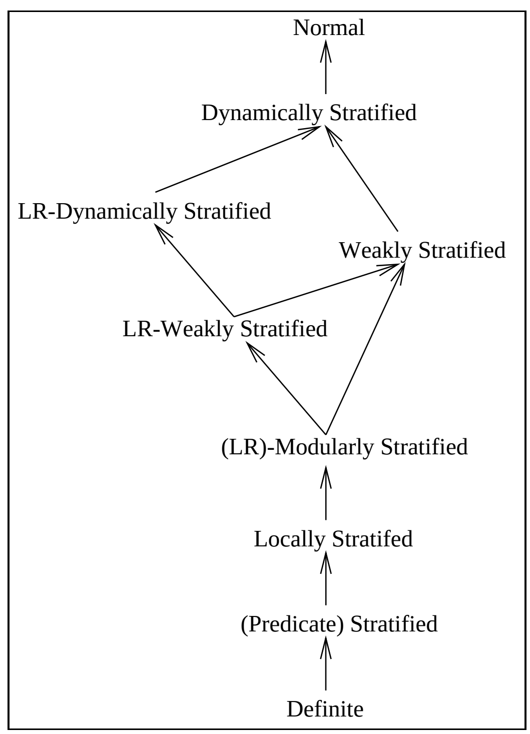


Figure 2: *Stratification Hierarchy*

Theorem 3.2.3 *The class of left-to-right dynamically stratified programs properly includes the class of left-to-right weakly, and (left-to-right) modularly stratified programs.*

3.3 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 [14] 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.

3.3.1 An Example of SLG_{Strat} Evaluation

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.

$$
\begin{array}{ll}
C_1: & \texttt{q(X) :- t(X), }\neg\texttt{q(X).} \\
C_2: & \texttt{q(a) :- }\neg\texttt{p, }\neg\texttt{r.} \\
C_3: & \texttt{t(a).} \\
C_4: & \texttt{p :- t(a), r.} \\
C_5: & \texttt{r :- p, }\neg\texttt{r, }\neg\texttt{q.} \\
C_6: & \texttt{q :- q(X).}
\end{array}
$$

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, which is necessary because the well-founded partial model of a program may 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. The following is the system after $q(a)$ is introduced:

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

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:

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

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 3.3.9).

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* [75] 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 $\neg\mathtt{p}$ 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:

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

Subgoals p and r are completed without any answers; by negation as failure literals $\neg\mathtt{p}$ and $\neg\mathtt{r}$ 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:

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

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 $q(a)$.

3.3.2 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 [8].

Definition 3.3.1 ([14]) *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.*
□

Notice that a computation may select a negative literal that is not ground. We try to solve a non-ground negative literal by negation as failure if it is sound to do so.

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

Definition 3.3.2 (*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 3.3.3 (*X-sequence*) [14] *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 3.3.4.

Definition 3.3.4 *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 $\rho_1 \cdot e$.* □

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 3.3.5 *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.* □

The intermediate states of the evaluation of a query are represented by a *system*.

Definition 3.3.6 (*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.* □

3.3.3 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 3.3.7 (*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 most general unifier 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 3.3.8 *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 3.3.9 (*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

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

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.

3.3.4 Derivation and SLG_{Strat} Resolution

Definition 3.3.10 *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 $\rho = \bigcup\{\rho' | (A : \rho') \in \mathcal{S}_j$ 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 3.3.11 *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 3.3.2, and the introduction of the NEGATIVE SUSPENSION transformation do not affect fundamental properties of SLG resolution.

Proposition 3.3.1 *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 3.3.1 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 3.3.12 (***Completed & 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 3.3.2 *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.*

Proof: Follows from the definitions of SLG_{Strat} transformations and Definition 3.3.12. ♢

3.3.5 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 $0 \leq i \leq n-1$. 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 3.3.3 *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.*

3.3.6 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 [14], 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 3.3.13 [14] *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 3.4.3), 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 3.3.4 *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 3.3.5 *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.*

$$I_{\mathcal{S}_i} \leq I_{\mathcal{S}_j} \text{ if } i \leq j.$$

Proof: Let $I_{\mathcal{S}_k} = \langle T_{\mathcal{S}_k}; F_{\mathcal{S}_k} \rangle$ for any k. The proposition follows immediately from the following two observations. First, once an answer is derived for a subgoal, that answer is never deleted by any transformation. So, $T_{\mathcal{S}_i} \subseteq T_{\mathcal{S}_j}$, for all $i \leq j$. Second, once a subgoal is completed no new answer can be added to that subgoal. In particular, if a subgoal fails, i.e. is completed without any answers, in a system $\mathcal{S}_i$, then it will also fail in $\mathcal{S}_j$, for all $i \leq j$. Thus, $F_{\mathcal{S}_i} \subseteq F_{\mathcal{S}_j}$. $\diamondsuit$

3.4 Correctness of SLG_{Strat} for Ground 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 3.2.5). All results of this section can be extended to any fixed-order computation rule. Proofs for theorems in this section are given in Appendix A.1

Lemma 3.4.1 *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 3.4.2 *Let P be a ground LRD-stratified program, Q be a ground atomic query, and $\mathcal{S}_0, \mathcal{S}_1, \ldots, \mathcal{S}_\alpha$ for some ordinal α 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 $\mathcal{S}$. Using the preceding lemmas the main result of this section, the correctness of SLG_{Strat} resolution for ground LRD-stratified programs, can be obtained.

Theorem 3.4.3 *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 $I_{\mathcal{S}} = M_P|_{\mathcal{S}}$.*

Proof: (Sketch) The proof is by transfinite induction on operations of the SLG_{Strat} evaluation. By Lemma 3.4.2 it is the case that no state in $\mathcal{E}$ will be undefined, so that there will be applicable transitions for all states until all subgoals are completed. Also note that all subgoals in the final system $\mathcal{S}$ will be completed. By Lemma 3.4.1 the truth value of completed subgoals will reflect that of the model of P. $\diamondsuit$

3.5 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 normal 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.*

3.5.1 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 3.2.1. 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 3.5.1 The following program is not LRD-stratified. As discussed in Example 3.1.1 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 3.5.2 The definition of `p/0` makes the following program non-stratified (and consequently non LRD-stratified).

```
a :- b.
a.
b :- ¬a, p.
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. ■

3.5.2 Towards Efficient Evaluation of the Well-Founded Semantics

By including the DELAYING, SIMPLIFICATION, and ANSWER COMPLETION transformations of SLG resolution (along with their necessary underpinnings),[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 Delaying*), 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 partial 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 3.5.1 *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

[2] We do not repeat their definitions here; they can be included as presented in [14].

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 3.5.1 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.

3.6 Applicability of Results

The results of Section 3.4 show that SLG_{Strat} can form the basis of an engine that evaluates fixed-order dynamically stratified programs, and the discussion of the previous section indicates that SLG_{Strat} can also be used as a basis to restrict the dynamic search in SLG-based engines that evaluate programs according to the well-founded semantics. Indeed, SLG_{Strat} and SLG_{RD} have been implemented in the SLG-WAM of XSB [58] versions 1.5.x and 1.6 on respectively. This engine is progressively described in the following three chapters of this dissertation. The performance results of Chapter 7 demonstrate that SLG_{Strat} and SLG_{RD} are indeed amenable to an efficient implementation.

The eventual goal of this dissertation is to evaluate the well-founded semantics of all normal logic programs in a search-efficient manner. Under that prism, LRD-stratified programs and SLG_{Strat} can be seen as only intermediate steps towards this direction. Despite that, the power of LRD-stratified programs over other stratification classes has proven of use in practical programs. For instance, it has been used in the underlying simplification algorithm of ***Semantica*** [36] which assigns meaning to natural language sentences. Furthermore, programs which may be fixed-order stratified according to a more static definition, such as modular stratification, may become left-to-right dynamically stratified when they are evaluated using a meta-interpreter, or when a simple program transformation that blurs the distinction between a predicate and an atom (such as one in the spirit of HiLog [11]) is applied to the program.

Even though the results in this chapter 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, the CORAL deductive database system [53] evaluates modularly stratified programs using the technique of Ordered Search [52]. Described simply, Ordered Search keeps a tree of dependencies and restricts magic evaluation to magic facts that are in an independent 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, consequently reducing the need for dynamic search. It appears possible in principle to extend these strategies to include early completion by checking whether a given magic fact is subsumed by an answer fact.

Chapter 4

The Abstract Machine for Definite Programs

4.1 Tabling for Definite Programs

Restricted to the class of definite programs, SLG (and thus SLG_{RD}) resolution reduces to SLD with *tabling* (also referred in the literature as tabulation or memoing), and does not significantly differ from other tabling formulations such as OLDT [71], Extension Tables [21], SLD-AL [77], or Backchain Iteration [79]. Tabling methods evaluate programs by keeping tables of subgoals and their answers, and by resolving repeated subgoals against *answer clauses* from the table rather than against program clauses. By using answer resolution in this manner, rather than repeatedly using program clause resolution as in SLD, SLG avoids looping and thus terminates for all programs with the *bounded term-size property.* In practice, not all predicates need to be tabled for termination or efficiency. For example, in the XSB system predicates may either be declared *tabled* in which case they are evaluated using SLG, or they may be implicitly assumed as *non-tabled* in which case SLD resolution is used for their evaluation.

When a subgoal of a tabled predicate is selected by the computation rule, a check must be made to determine whether the subgoal is redundant or not. In the case of the XSB system, this currently amounts to a *variant check* of whether the new subgoal is a variant of one that already exists in the *subgoal table.*[1] If not, the subgoal is termed *generator*, it is entered into the table, and program clause resolution is used to generate the answers of this subgoal. An *answer table* containing a set of answers is associated with each tabled subgoal. If the subgoal already exists in the table, the subgoal is termed *active* and answers from its answer table are resolved against it.

We introduce the operations that are involved in tabled-based evaluation of definite

[1] In logic, one atom is a *variant* of another if the two atoms are the same up to variable renaming. Tabling systems may perform variant checks as XSB currently does, or subsumption checks, where an atom A_1 *subsumes* an atom A_2 if there is some substitution θ which unifies A_1 and A_2 such that the domain of θ includes only variables of A_1.

programs through the following concrete example.

Example 4.1.1 Consider the evaluation of the query `?- p(a,Z)` with respect to the program in Figure 3. We assume, as we do throughout this document, a leftmost literal selection strategy. The declaration `:- table p/2` indicates that SLG resolution is to be used for calls to predicate `p/2`. The forest of SLG trees for this example along with its table at the end of the evaluation is depicted in Figure 3. Nodes of the trees in the forest consist of annotated SLG clauses: a clause with status *status* of an SLG tree rooted by subgoal S is depicted as S : *status* : A :- *Body*, where the head of the clause, A, is an *answer template* that accumulates substitutions for the variables of the subgoal. We use the term *SLG system* to refer to an SLG-forest and the associated subgoal and answer tables. SLG systems can be seen as capturing the *states* of an *SLG evaluation* of a query against a program.

Let us examine elements of the SLG forest in detail. A new SLG tree is added whenever a call to a tabled subgoal is made that is not a variant of any previous tabled subgoal. The addition of the new tree in the system is performed by the NEW SUBGOAL operation. The evaluation of query `p(a,Z)` begins by creating a new tree rooted by `p(a,Z)` and by using program clause resolution to produce answers for this subgoal. The new subgoal is first resolved against the clause `p(X,Z) :- p(X,Y),p(Y,Z)`, creating node 1 in Figure 3 with status *active*. In node 1, the selected literal `p(a,Y)` is tabled, so the NEW SUBGOAL operation is again applicable. Since `p(a,Y)` is a variant of a tabled subgoal that was previously encountered, no new tree is created; instead, the execution of this clause *suspends* and waits for the generation of answers for this subgoal. If (all) answers for this subgoal were already present, ANSWER RETURN operations could be immediately scheduled to return them. However, since this subgoal currently has no answers, the only operation that applies is to node 0, and the second program clause is tried. This resolves the subgoal `p(a,Z)` against the program clause `p(X,Z) :- e(X,Z),q(Z)`, producing node 2 of this tree. Since the selected literal for node 2 is of a predicate that is not tabled, a NEW INTERIOR operation is applied to this node and Prolog-style program clause resolution is used to resolve this literal. Depth-first resolution of non-tabled nodes continues until node 4 which contains no further literals to resolve. The NEW ANSWER operation changes the status of this node to *answer*, and the bindings in the *answer template* of node 4 form an answer of subgoal `p(a,Z)`. Since this answer, `p(a,b)`, is not a variant of any other answer node of this subgoal, it will be inserted into `p(a,Z)`'s answer table. When all possible program clause resolution has been performed, this new answer will be scheduled to be returned, via the ANSWER RETURN operation, to all active nodes whose selected subgoal is a variant of `p(a,Z)`. In our example, the only such active node is node 1. After Prolog-style program clause resolution is exhausted (node 6) for the subtree rooted at node 2, answer clause resolution will create node 7, which in its turn might produce other new subgoals and answers both for new subgoals and for existing one(s).

In our example, through answer clause resolution, the evaluation will eventually give rise to two other tabled subgoals, `p(b,Z)` and `p(c,Z)`, each of which will be recorded in the global table and form the root of its own SLG tree. These subgoals are shown enclosed in

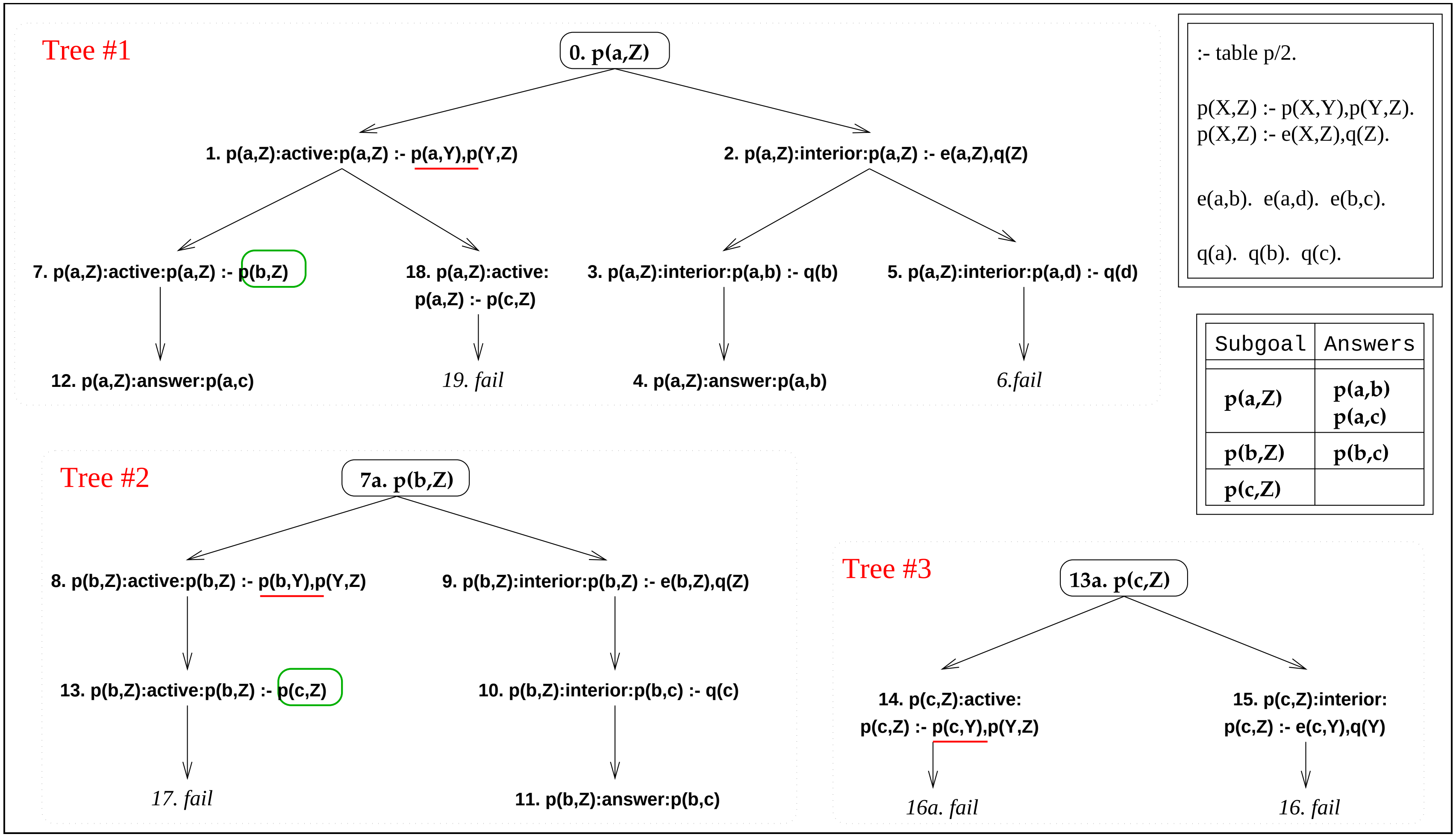


Figure 3: *Program, SLG Forest, and Tables for the query* `?- p(a,Z).`

oval boxes in the SLG trees of Figure 3. The number associated with each of the nodes in Figure 3 corresponds to the order of their generation by the SLG-WAM. It can be seen that the evaluation is largely depth-first, although the children of nodes 1, 8, and 14 — nodes 7 and 18, 13, and 16a — form an exception, caused by the need to suspend these computations in order to produce answers for the selected subgoals of these nodes (subgoals that need to be suspended are shown underlined in Figure 3). In general, the process of expanding nodes, adding new answers and returning them to consuming subgoals, continues until all possible program and answer clause resolution has been performed for all mutually dependent sets of tabled subgoals, called *components* of the SLG system. If no further resolution steps are possible for the selected literals in a component, its subgoals are *completely evaluated*. In the SLG system of this example there are no mutual dependencies among subgoals, and so there are three singleton components {`p(a,Z)`}, {`p(b,Z)`}, and {`p(c,Z)`}. Using the COMPLETION operation *incrementally* (not shown in Figure 3), trees for a component can be disposed as soon as they are determined to be completely evaluated. In our example, the latter two components could be completed after nodes 16 and 17 are produced. The completion of subgoal `p(c,Z)` with no answers allows an immediate failure of the selected literal of node 18 of the forest. The selected literal of this node does not need to be suspended since it is now known that answers for `p(c,Z)` cannot be produced. After node 19 has been created, {`p(a,Z)`} is also completely evaluated, and its tree could also be disposed. ■

In summary, tabled evaluation of definite programs uses the following five operations:

$\boxed{\text{NEW SUBGOAL}}$ that creates either a *generator* or an *active* node N for a tabled subgoal S. If S is not present in the evaluation (N is a generator), creates a new SLG tree with root S and children generated by program clause resolution against S. Used in nodes 0, 7, and 13.

$\boxed{\text{NEW ANSWER}}$ that adds an answer A of a tabled subgoal S to the answer table of S if A is not present, and returns this answer to the *generator* node of S. Used in nodes 4, 11 (creating node 12), and 12.

$\boxed{\text{ANSWER RETURN}}$ that returns an answer A of a tabled subgoal to *active* nodes of that subgoal. Used in nodes 1 (creating nodes 7 and 18), in node 8 (creating node 13).

$\boxed{\text{COMPLETION}}$ that marks as *completed* sets of subgoals when it is determined that all possible answers have been derived for them. Used for the sets {`p(c,Z)`}, {`p(b,Z)`}, and {`p(a,Z)`} after nodes 16, 17, and 19 are produced.

$\boxed{\text{NEW INTERIOR}}$ that resolves selected subgoals of non-tabled predicates using Prolog-style SLD resolution. Used in nodes 2, 3, 5, 9, and 10.

4.1.1 Definite SLG-WAM: Overview

As mentioned, in SLG evaluation of definite programs certain tabled subgoals resolve against answers rather than against program clauses. The need to perform answer clause resolution makes the tabling operations *asynchronous* processes. As shown by Example 4.1.1, tabling presents an asynchrony between the generation of subgoals and answers and the consumption of answers by active subgoals: more than one instance of tabled a subgoal may be encountered before the evaluation is able to generate answers for these subgoals. Furthermore, when tabling is used in conjunction with a non-deterministic relational language, such as Prolog, this asynchrony is even more apparent. For example, note that the computation rule may select a tabled subgoal that currently either has no answers, or contains only some of its answers, or is such that all of its answers have already been derived.

This asynchrony between tabling operations introduces an additional level of non-determinism that the evaluation strategy should handle. Given an SLG system, i.e. a particular state of an SLG derivation, there might be various SLG operations that are applicable. SLG resolution allows the use of an arbitrary scheduling strategy for selecting the operation to apply. Even though this degree of freedom in a resolution method is theoretically appealing, an abstract machine for SLG has to be deterministic; i.e. it has to fix *a priori* the scheduling strategy that is going to employ. Naturally, different scheduling strategies have different properties; for example they might have different performance characteristics, or even influence parts of the abstract machine's architecture. We discuss issues related to scheduling of SLG operations in Section 4.4.

Independently, however, of the scheduling strategy that is chosen, an abstract machine for SLG resolution has to be able to *suspend* computations when encountering repeated instances of tabled subgoals which currently have no answers. Furthermore, these active subgoals must be retained at least until the computation reaches a fixpoint: until all applicable answers have been derived and resolved against the subgoals. The branches of the search trees corresponding to the suspended subgoals must either be maintained or reconstructed. Likewise, newly derived answers require scheduling and the ability to *resume* suspended computations. The following section describes extensions to the WAM to provide these functionalities that, as previously mentioned, are independent of the scheduling strategy that is being employed.

4.2 Suspending and Resuming Computations

In general, tabled evaluation of logic programs requires that computations *suspend* when *active* subgoals have exhausted the set of answers currently in the table, so that computation may proceed to eventually produce new answers. With respect to Example 4.1.1, computation must suspend in nodes 1, 8, and 14. Within the WAM framework just suspending a computation is easy: it can simply be done by failing to the previous choice point (without reclaiming any stack space). Naturally, this is not enough. Computations are suspended with the intent that they will be later *resumed* (reactivated) when answers are derived for the suspended subgoals. Thus, SLG-WAM mechanisms that implement *suspend* and *resume* have to co-operate.

Described informally, the *suspend* operation records the state of the abstract machine (e.g., information about the contents of the SLG-WAM stacks, and the success continuation of the suspended computation). On *resume*, this saved state gets restored.

For most stacks, saving just the value of the corresponding WAM register is sufficient information to restore the stack's state. However, the restoration is only possible provided that new computations will not overwrite information created before the suspension.

4.2.1 Preserving Environments through Freeze Registers

To ensure that information about the state of the machine is not lost for suspended computations, WAM stacks are frozen and, assuming that stacks grow up,[2] space is retained below the *freeze registers*. In definite programs, computations suspend and stacks are frozen whenever an *active* subgoal is encountered; this is done by the `tabletry` SLG-WAM instruction which will be presented in Section 4.5. Figure 4 shows states of the choice point and the completion stack (a new stack that the SLG-WAM adds to the WAM) while executing the program of Example 4.1.1. Note that on encountering the active subgoal `p(a,Y)` in node 1, the computation gets suspended, a freeze point is set (denoted as *freeze1* in Figure 4(a)), computation continues with node 2 of the tree, and the next choice point (for `e(a,Z)`) is allocated above this frozen segment of the choice point stack.

The introduction of freeze registers affects the WAM `try` and `trust` instructions; to record new information in the choice point stack the maximum of **B** register, and the choice point freeze register (**BF** register) has to be taken into account. For local environments it affects the `allocate` instruction which must check for the greatest of the environment register (**E** register), the environment backtrack register (**EB** register), and the environment freeze register (**EF** register). Also, freeze registers turn the trail stack into a tree and allocation of new trail entries requires a check of the trail freeze register (**TRF** register), as well as the usual WAM trail register (**TR** register).[3]

The heap also has its freeze register, **HF** register, which also turns it into a tree. To avoid the significant overhead of having to check two registers at every write to the heap, upon backtracking and execution of a `trust` instruction, the **H** register is set to the greatest of **HB** register and **HF** register rather than to **HB** register as in the WAM. Instructions can then build information on the heap just as in the WAM.

Frozen segments in the stacks can be deallocated only at the point when it is known that

[2]Throughout this document all stacks are shown growing upwards. However, frames are depicted with their first cell on top and their last on the bottom.

[3]The WAM checks the **EB** and **HB** registers to determine whether a binding is conditional and its variable needs to be trailed. The SLG-WAM must also check the corresponding freeze registers, **EF** and **HF** registers, in this situation.

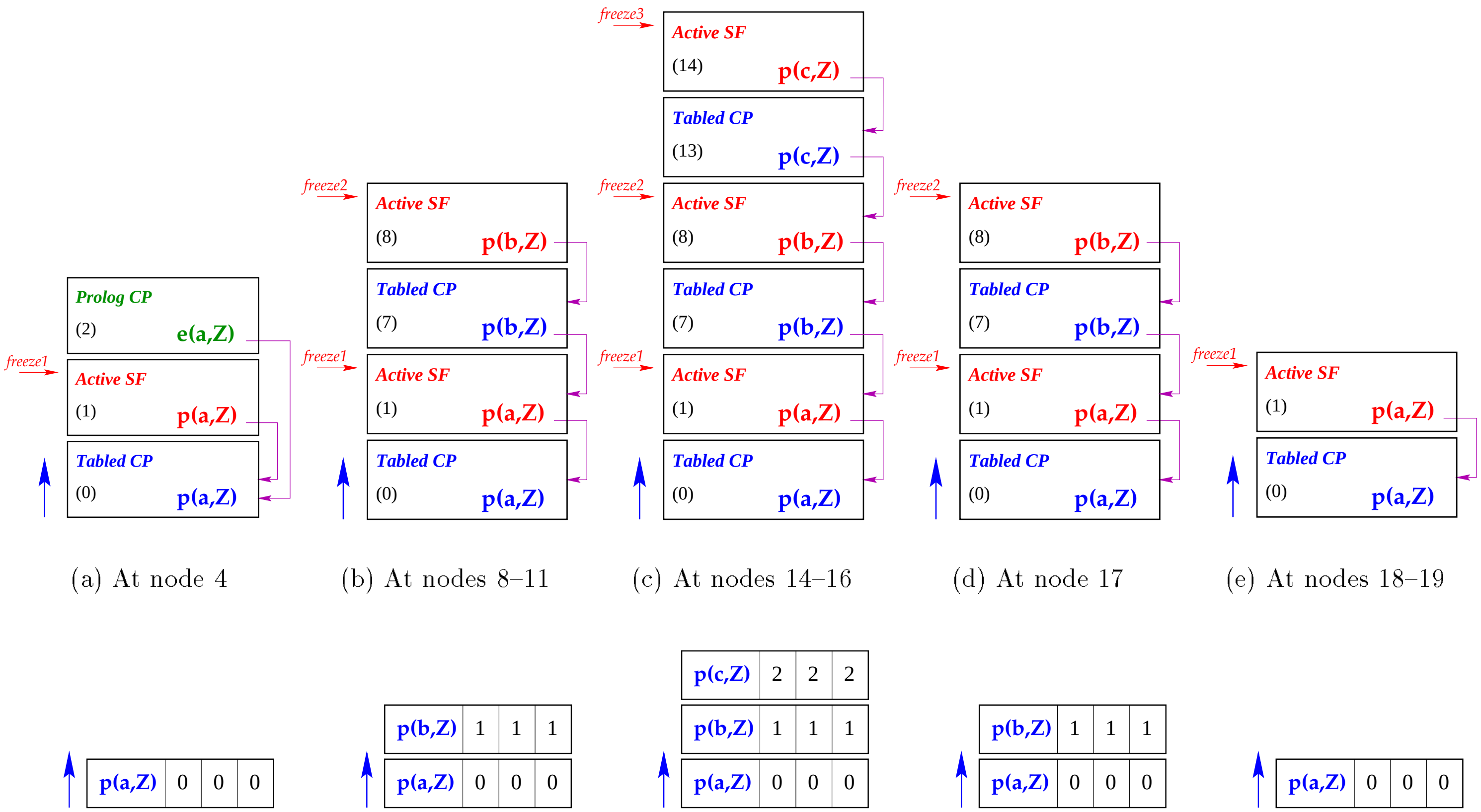


Figure 4: *Choice Point and Completion Stack States for Program of Figure 3.*

no more *resume* operations can occur for the corresponding set of suspended computations. This condition holds only after the generation and consumption of all possible answers for a set of subgoals; i.e. only when it can be determined that this set of subgoals is *completed*. Reclaiming frozen stack space is a major reason for the existence of the COMPLETION operation for definite programs (an issue that is further discussed in Section 4.6).

4.2.2 Forward Trail

The *trail* in the WAM is organized as an array of addresses of (local stack or heap) variables which must be reset to unbound upon backtracking. For Prolog-style execution the trail works as a stack and a global register (the **TR** register) always points to the top of the trail stack. Only *conditional bindings* (i.e. those that affect the binding of a variable existing before the creation of the current choice point) need to be trailed; a property of the WAM that continues to hold in the SLG-WAM.

However, for tabled-based evaluation of logic programs, the trail requires modification to execute the ANSWER RETURN operation, which returns answers to suspended consuming subgoals. To do so, the variable bindings of *suspended active* subgoals have to be restored; this is done using a *forward trail* mechanism (which was first introduced in [81]). Forward trail frames have the format shown in Figure 5. In addition to containing the address of

PrevTrail	Pointer to previous trail frame
VarValue	Value to which the variable was bound
VarAddr	Address of the trailed variable

Figure 5: *Format of (Forward) Trail Frames.*

a trailed variable as in the WAM, they contain information about the value to which the variable was bound. Furthermore, as the trail is also a tree rather than a stack, each trail frame has to maintain an explicit pointer to the previous trail frame (using its *PrevTrail* field). The overhead that the forward trail incurs to the WAM is measured in [69].

4.2.3 Resuming Suspended Computations by Restoring Environments

On *resume*, the values of the WAM registers at the time that the computation was suspended (as saved using an *active subgoal frame* which constitutes the environment of an active subgoal) are restored and execution continues with the success continuation of the suspended computation.

At any point in time, the state of the heap and the local stack reflects the variable bindings that correspond to a certain point in the search tree. To properly resume computations that have been suspended on the derivation of answers, these bindings must also be restored through a restore_bindings operation. The algorithm that implements this operation is shown in Figure 6 and its details are described below.

```
Algorithm restore_bindings(old_breg, new_breg)
    start_trreg := choice_point_trreg(old_breg);
    end_trreg := choice_point_trreg(new_breg);
    while (start_trreg != end_trreg)
        while (start_trreg > end_trreg)
            untrail(trail_var(start_trreg));
            start_trreg := trail_parent(start_trreg);
        while (end_trreg > start_trreg)
            end_trreg := trail_parent(end_trreg);
    end_trreg := choice_point_trreg(tbreg);
    while (end_trreg > start_trreg)
        * trail_var(end_trreg) := trail_val(end_trreg);
        end_trreg := trail_parent(end_trreg);
```

Figure 6: *The* restore_bindings *operation.*

Using the information in the forward trail, the restore_bindings operation restores the bindings of the suspended environments so that they correspond to those of a different node in the SLG search tree. It starts from the tree node specified by *old_breg*, and switches to the variable bindings of the node corresponding to *new_breg*. Both *start_trreg* and *end_trreg* follow their parent chains until a common ancestor is reached, with *start_trreg* untrailing as it goes. Then, variables on the path from *end_trreg* to the common ancestor are rebound. Note that since, among other functions, choice points are used to reconstitute environments, each tabled predicate requires a choice point, even if the predicate is defined by a single clause.

4.3 Architecture and Data Structures

The SLG-WAM extends the WAM memory layout both to include a representation of subgoal and answer tables, to properly execute answer resolution, and to operate over a forest of SLG trees rather than a over a single SLD tree. The SLG-WAM contains the usual memory areas of the WAM: *local* and *global stack*, *choice point stack*, *trail* and *unification stack*. In addition to using a forward trail and turning these WAM areas into trees by introducing freeze registers, the SLG-WAM adds two further memory areas: a *Table Space* and a *Completion Stack*. The *Table Space* stores tabled subgoals and their answers. The *Completion Stack* is used to detect when a set of subgoals has been completely evaluated and is described in Section 4.6.

To stay within the WAM paradigm, it is desirable for the various trees in the SLG forest to share the same stacks. A closer approximation of the SLG-WAM's memory layout is as a single *SLG search tree* which can be constructed by inserting each tree with root goal G as a subtree below the first node N_G whose selected literal is G, and rewriting the variables of G so that they are shared with N_G. Figure 7 illustrates how the SLG search tree for the program of Example 4.1.1 can be constructed from the forest of SLG trees shown in Figure 3. Only nodes 0–17 are shown in Figure 7.

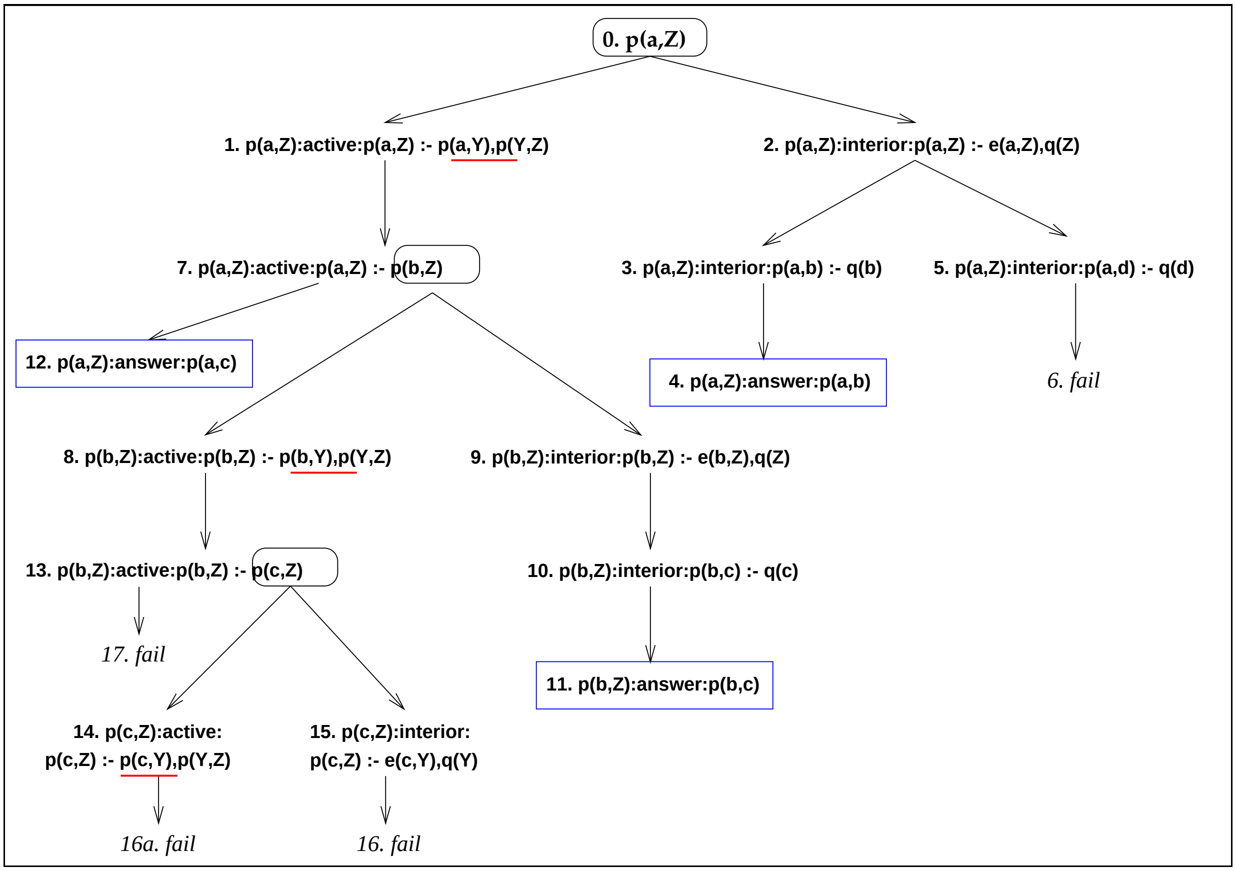


Figure 7: *The SLG Forest of Figure 3 as a single SLG Search Tree.*

4.3.1 Choice Points

As mentioned in Section 4.2.1, new choice points allocate their frames on top of the choice point stack by comparing the values of the **B** and the **BF** register. Choice point information for backtracking (either to program or answer clauses) is derived through the **B** register, as in the WAM. There is however, one major difference with the WAM: the choice point stack rather than containing only one branch of the computation, keeps information about several branches of the search tree. In the WAM, the parent of a given choice point is the previous frame in the choice point stack. In the SLG-WAM, a given choice point may have been allocated above the freeze register so its *parent choice point* may lie arbitrarily deeper in the stack. For example, note the choice point stack in Figure 4(a). Due to the suspended computation of the active subgoal for `p(a,Z)`, the previous choice point of the (Prolog) choice point for `e(a,Z)` is the (generator) choice point for `p(a,Z)`. Thus, choice point frames must maintain explicit pointers to their previous choice points, as do the environment frames of the local stack in the WAM.

The format of a *generator choice point frame* of the definite engine, which is used to execute program clause resolution for tabled subgoals, is depicted in Figure 8. It is shown as being split into three parts. The topmost part is subgoal-dependent and contains the argument registers of the subgoal. The middle part can be thought as an "elementary choice point frame" of the abstract machine for tabling; it contains information that should be restored on backtracking regardless of whether the choice point is for a tabled subgoal or for a subgoal which is evaluated using SLD resolution. In addition to the usual cells of a WAM choice point, it contains an explicit pointer to the previous choice point in the search tree. Finally, the last part is specific to generator choice points. It contains a pointer to the table entry associated with that choice point (the *Subgoal Frame*), and the values of the freeze registers at the time of choice point creation. Even though creating a generator choice point frame does not require freezing the stacks, the values of the freeze registers must be recorded so that they can be properly reset when the subgoal associated with this choice point is completed. In Figure 8, components that are not part of regular WAM choice points are marked with an asterisk.

A_n	Argument Register n
$\vdots$	$\vdots$
A_1	Argument Register 1
FailCont	The Failure Continuation
EBreg	Environment Backtrack Point
Hreg	Top of Global Stack (Heap)
TRreg	Top of Trail
CPreg	Return point of call to predicate
Ereg	Parent Environment
*PrevCP**	Previous Choice Point
*Subgoal**	Pointer to the Subgoal Frame
*BFreg**	Choice Point Freeze Register
*HFreg**	Heap Freeze Register
*TRFreg**	Trail Freeze Register
*EFreg**	Local Stack Freeze Register

Figure 8: *Generator (Tabled) Choice Point.*

4.3.2 Table Space

The *Table Space* records information about tabled subgoals and their answers. In a structure copying framework, subgoals and answers have to be copied into a separate memory area so that (i) variables in subgoals and answers are disconnected from variables in program clauses, and (ii) information about subgoals and answers survives the effects of backtracking and possible space reclamation (i.e. it is *persistent*). We note that structure sharing implementations of tabling are also possible — indeed, [67] and [78] effectively describe structure sharing algorithms for logic programming languages based on tabling. While useful bounds can be derived for the amount of copying needed by a structure-sharing approach, such approaches usually lead to complicated implementation mechanisms, may be subject to high constant overheads, and in any case do not appear suitable for a WAM-based implementation.

Elements of the table space are created dynamically and may need to be repeatedly accessed during the course of SLG evaluation. Subgoals and answers have to be checked for existence and inserted in the appropriate tables if they are newly encountered (*subgoal* and *answer check/insert* operations in the terminology of [50]). Furthermore, answer resolution requires that answers are copied out of their tables when returned to their consuming subgoals. Consequently, the design and implementation of data structures and algorithms for high-performance table space access is a central issue for an abstract machine based on tabling. The table space can be implemented using any data structure for term indexing (such as hashing, tries, or discrimination nets). The current version of XSB uses *tries* as the basis for tables. Tries provide complete discrimination, and permit a lookup and possible insertion to be performed in a single pass through a term. In this chapter, we provide only a brief description of the layout of the table space using tries; full details are presented in [50].

The table space for Example 4.1.1 is represented schematically in Figure 9.[4] The instruction at the entry point of each tabled predicate P points to a *table information pointer* or *TIP* (details of the SLG-WAM instruction set will be given in Section 4.5). The *TIP*, in turn, points to a *subgoal trie* index structure which stores the subgoals of predicate P. We note that *TIPs* are persistent data structures, as opposed to *subgoal tries* which are created and abolished dynamically.

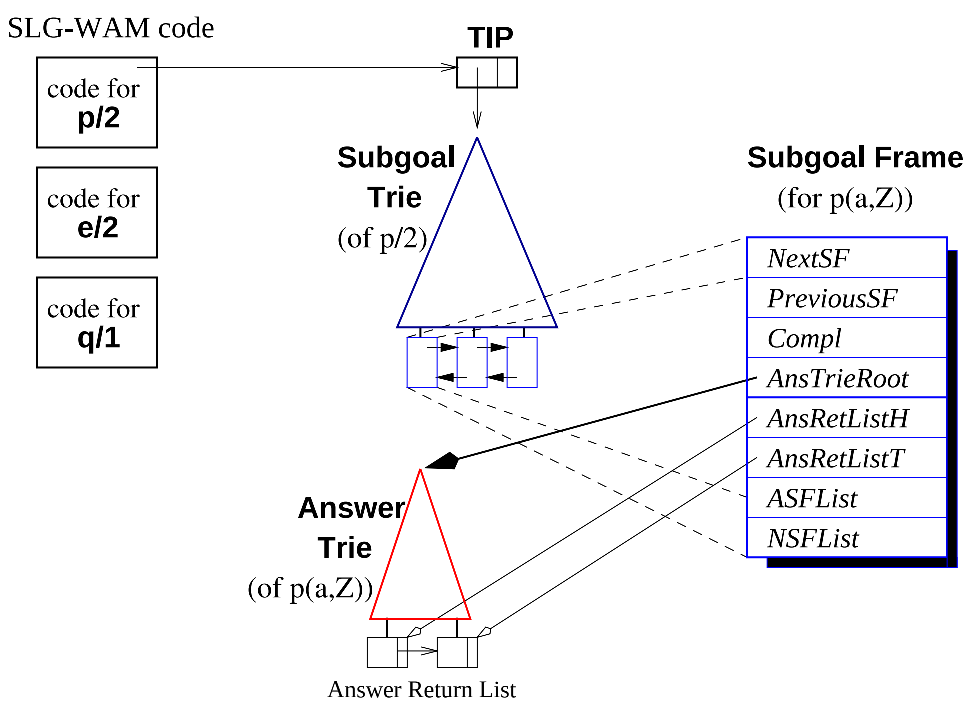


Figure 9: *Relationships between elements of the Table Space.*

Subgoal Frames Each root-to-leaf path through a subgoal trie corresponds to a single subgoal. Leaf nodes of the subgoal trie point to *subgoal frames*. Subgoal frames (whose

[4]The answer trie of the `p(b,Z)` subgoal is not shown in Figure 9.

format is shown in Figure 10) point to the root of the associated *answer trie*, and to an associated *completion stack frame*. Subgoal frames also contain pointers to the following

NextSF	Next Subgoal Frame
PreviousSF	Previous Subgoal Frame
Compl	Associated Completion Stack Frame
AnsTrieRoot	Root of the Answer Trie
AnsRetListH	Head of the Answer Return List
AnsRetListT	Tail of the Answer Return List
ASFList	List of Active Subgoal Frames
NSFList	List of Negation Suspension Frames

Figure 10: *Format of Subgoal Frames.*

three lists: 1) a list of (variant) active subgoals that are suspended on the generation of new answers (*ASFList*), 2) a list of (variant) instances of the subgoal that are suspended till the completion of the subgoal (this list containing *negation suspension frames* is used for programs containing negation), and 3) a list of answers for the subgoal (for efficiency, pointers to both the head and the tail of this list are maintained). Information about these three lists is not needed once the subgoal is completed; consequently the bottom part of Figure 10 can be reclaimed upon completion. Alternatively, it can be part of the associated completion stack frame that is automatically reclaimed upon the completion of the subgoal. To facilitate insertion and deletion, subgoal frames are maintained in a doubly linked list.

The answers of each tabled subgoal have their own trie which differs slightly from that of the subgoals, because answers need to be accessed in a way that subgoals are not. When an answer is derived, it must be checked as to whether it is already in the table. This is similar to the redundant subgoal check, and tries efficiently support this operation. However, answers also need to be returned to all consuming instances of the subgoal. Since the generation and consumption of answers are asynchronous operations, and new answers may be inserted *anywhere* in a trie, it is not possible to perform the ANSWER RETURN operation by sequentially backtracking through an answer trie of an *incomplete* subgoal starting from its root. Therefore, an explicit list of answers (uniquely identified by leaf nodes of the answer trie), has to be maintained. By maintaining the *answer return list* in an order which reflects the creation times of its members, we guarantee that no answer is returned to the same active subgoal more than once.

Active Subgoal Frames Repeated instances of tabled subgoals are kept in *active subgoal frames* which, like generator choice points, are stored on the choice point stack. *Active subgoal frames*, serve as environments into which answers can be returned by means of an ANSWER RETURN operation. *Active subgoal frames* have essentially the same structure as generator choice point frames except that that they have no need for freeze registers. We also note that the *substitution factoring* operation (discussed on page 53) allows active subgoal frames to contain only the *variables* of argument registers rather than the argument registers themselves. Each active subgoal frame contains a pointer, *LastAnswer*, to the last answer of

FailCont	The Failure Continuation
EBreg	Environment Backtrack Point
Hreg	Top of Global Stack (Heap)
TRreg	Top of Trail
CPreg	Return point of call to predicate
Ereg	Parent Environment
*PrevCP**	Previous Choice Point
Subgoal	Pointer to the Subgoal Frame
LastAnswer	Last Answer that was Consumed
PrevASF	Previous Active Subgoal Frame

Figure 11: *Format of Active Subgoal Frames.*

the *answer return list* that the active subgoal frame has consumed. They also have a field, *PrevASF*, that points to the previously encountered active instance of the same subgoal. Using this field, all active instances of a tabled subgoal are effectively chained in a list (this list is the place where the *ASFList* field of subgoal frames points to). Furthermore, depending on the scheduling strategy that is being employed (an issue that is discussed in the following section) they may or may not need a *PrevCP* field, which points to the previous choice point in the search tree. These pointers allow the active subgoal frame to do double duty. The *PrevCP* pointer allows it to be used as choice point for the `answer_return` instruction which returns answers to the active subgoals. When backtracking exhausts the answers in the answer return list, the **B** register is set to that saved in the *PrevCP* field. The pointer to the previous active subgoal is used by the `answer_return` instruction for backtracking through the chain of suspended active subgoals, until the chain is exhausted (see Figure 17).

4.4 On Scheduling

In tabled-based evaluation of a relational language there is an intrinsic asynchrony between the generation and the consumption of answers by instances of tabled subgoals. On encountering repeated instances of a subgoal that currently has no answers, the scheduling decision is unique: the subgoals and the associated computations have to suspend. Consider, however, the various scheduling choices on the derivation of an answer for a tabled subgoal S: the engine may return this answer to all currently known instances of S, to only some of them, or it might postpone the ANSWER RETURN operation for a later point of the SLG evaluation. Similarly, on encountering an active instance of a (still incomplete) subgoal for which some of its answers have already been derived, the engine can choose to immediately return the current set of answers, or suspend the subgoal anyway (as it would have to do in the situation where the subgoal had no answers). Clearly, other scenarios where multiple scheduling alternatives exist are also possible.

Due to this asynchrony, implementations of tabling have to choose a scheduling strategy:[5] i.e. they have to decide when (and how) to perform the ANSWER RETURN operation. Naturally, different scheduling strategies have different characteristics and may result in tabled evaluations with different properties. Chief among them is their time and space performance. Freire, Swift, and Warren in [27] discuss various scheduling strategies and present their detailed experimental evaluation.

For the purposes of this document, that describes the general architecture of the SLG-WAM, we use a scheduling strategy that allows for a tight correspondence between the tabling instructions that we present in the next section, and the SLG operations of Section 4.1. More specifically, this scheduling follows the rules described below:

Scheduling Algorithm 1 *Rules of the scheduling strategy.*

i. The first occurrence of a tabled subgoal (the *generator*) is used for program clause resolution and, because of rule (iv), this instance of a tabled subgoal does not need to be suspended.

ii. Subsequent occurrences of *incomplete* subgoals are always suspended (regardless of whether the subgoals have answers or not, when the subgoal instance is encountered).

iii. On encountering an instance I of a tabled subgoal S that is *complete* (i.e. a subgoal such that all of its answers have already been generated), the computation is not suspended. Instead, the answers of S are immediately returned to I by backtracking through the answer trie (using the instruction set presented in Section 4.5.3).

iv. On deriving a new answer for a subgoal S, the answer is recorded in the answer table of S, and it is immediately returned to the *generator* instance of S, effectively performing a *first call optimization*.

v. On backtracking to a generator choice point for a (still incomplete) subgoal S, answers are scheduled to be returned to all currently known *active* instances of S.

vi. A subgoal is not completed until all its answers have been returned to all its *active* instances.

This scheduling strategy can be thought as a series of iterations where answers are returned to active instances of a set of subgoals Σ until a fixpoint is reached for this set. The fixpoint has been reached when no more answers have been derived for subgoals in Σ. The combination of this fixpoint condition with rule (vi) ensures the correctness of the strategy. In general, scheduling strategies have to guarantee that all answers in the table are returned

[5]The term "scheduling strategy" does not imply that the decision should be global: the scheduling may be predicate-based or even atom-based. However, since the instruction set and, to a lesser extent, the data structures depend on the scheduling, the different scheduling choices that the abstract machine can make have to be fixed *a priori*.

(preferably not more than once) to all instances of tabled subgoals. This requires some sort of bookkeeping of places where answers have already been returned.

In terms of implementation, the choice point stack serves as a scheduling stack both for performing program clause resolution and for returning answers. Furthermore, each active subgoal keeps track of the set of answers it has consumed, by maintaining a pointer to the last consumed answer substitution in the answer return list. The fixpoint check is performed as part of the COMPLETION operation: active nodes are checked whether the answer substitution which they last consumed is the currently last element of the answer return list (i.e. whether they have consumed all answers that have been derived).

A slightly optimized variant of this scheduling has been experimentally shown to be efficient in terms of minimizing restoration of suspended environments and choice point creation and, in fact, is the scheduling strategy that is currently used in the XSB system. We indicate this optimization in Section 4.5.2. Further details can be found in [61, 27].

4.5 Extending the Abstract Machine Instruction Set

Besides extending the WAM data structures, the SLG-WAM also adds a set of new instructions to the WAM which we present in this section in two steps: we first try to motivate an initial instruction set design by relating its members to the SLG operations for definite programs that were presented in Section 4.1, and we then optimize this initial instruction set.

We also mention that answers from completed tables can be returned by sequentially backtracking through them; this operation can always be performed by interpreting the answer table data structure. However, answer tables based on tries naturally lend themselves to a dynamic compilation scheme. *Compiled tries* (tries annotated with WAM-like instructions) allow backtracking through answers to occur at speeds comparable to compiled WAM code. Details of this set of *trie instructions*, which is completely orthogonal to the above mentioned extension of the WAM instruction set, are presented in Section 4.5.3.

4.5.1 A Reconstruction of the Basic Instruction Set for Tabling

We motivate the tabling instructions by considering the instructions that need to be generated for a tabled predicate $\mathtt{t}/n$ defined by k clauses. Using the program transformation shown below, all possible indexing code of the WAM is pushed one level down to a *new* pred-

`:- table t/`n.	$\longrightarrow$	`:- table t/`n.
`t(`$t_{1,1},\ldots,t_{1,n}$`) :- `B_1.		`t(`$X_1,\ldots,X_n$`) :- p(`$X_1,\ldots,X_n$`).`
$\vdots$		`p(`$t_{1,1},\ldots,t_{1,n}$`) :- `B_1.
`t(`$t_{k,1},\ldots,t_{k,n}$`) :- `B_k.		$\vdots$
		`p(`$t_{k,1},\ldots,t_{k,n}$`) :- `B_k.

icate `p/`n which is evaluated using Prolog-style resolution. Notice that these two programs are equivalent with respect to observables.

The Prolog predicate `p/`n can be compiled using the instruction set of the WAM [80]. We concentrate on the instructions needed for the tabled evaluation of predicate `t/`n defined by the single rule:

$$\mathtt{t}(X_1, \ldots, X_n) \coloneq \mathtt{p}(X_1, \ldots, X_n).$$

Tabled predicates defined by such chain rules could be thought as compiled in a "WAM-like fashion" as shown in Figure 12. To a WAM connoisseur, this WAM-like code may seem

L_1 :	try_me_else	L_7	
L_2 :	new_subgoal_check_insert	n	TIP
L_3 :	allocate		
L_4 :	call	1	`p/`n
L_5 :	new_answer_check_insert	n	v_1
L_6 :	deallocate		
L_7 :	answer_return_loop		
L_8 :	completion_check		
L_9 :	trust_me		

Figure 12: *WAM-like code for tabled predicates.*

strange. Indeed, there are various points that require explanation.

Roughly, the first part of this pseudo-WAM code creates instances of tabled subgoals, records these subgoals and their answers into the table space, derives answers for these subgoals by performing program clause resolution, records these answers into the table space, and returns them to the *generator*. The second part performs answer clause resolution against *active* instances (until the fixpoint condition is satisfied), and checks subgoals for completion. Notice how this pseudo-WAM code reflects — and depends on — the scheduling strategy that is being employed.

The four pseudo-instructions that are new to the WAM, namely, new_subgoal_check_insert, new_answer_check_insert, answer_return_loop, and completion_check correspond to the SLG operations NEW SUBGOAL, NEW ANSWER, ANSWER RETURN, and COMPLETION, respectively. The procedures that implement these pseudo-instructions are to a large extent interpretive since they rely on information that is *dynamic in nature* (checking whether a particular subgoal or answer is new or already exists in the tables, whether all answers have been returned to all subgoal instances, and whether it can be determined that subgoals are complete).

Finally, note that contrary to the WAM, resolution against the program clause for `t/`n requires both a choice point and a local environment even though `t/`n is defined by a single clause with no *permanent* variables.

- Choice point creation is necessary since the use of the completion_check effectively adds an extra clause to each tabled predicate. Furthermore, recall that among other functions, choice points of tabled predicates are used to reconstitute environments of suspended computations (in the way explained in Section 4.2.3). This use of choice points also explains the unorthodox placement of the trust_me instruction (that removes this choice point) *after* the completion_check instruction. The later instruction does not succeed until the completion of the subgoal is determined (cf. Figure 24).

- The local environment is required by the new_answer_check_insert instruction. A special cell of the environment (which can be thought of as an extra permanent variable — denoted as v_1 in Figure 12)[6] holds a pointer to the generator choice point, which contains both the information used for the answer check/insert operation and a pointer to the subgoal frame and by extension to the answer trie (cf. Figure 9). This extra cell of the local environment is needed to record answers in the answer table of the subgoal; thus, the deallocate instruction has to occur *after* the new_answer_check_insert instruction. Consequently, the *last call optimization* is not applicable to tabled predicates; other optimizations like *environment trimming*, however, can be applied.

4.5.2 Optimizing the Basic Instruction Set for Tabling

Note that with the instruction set presented in the previous section, tabled predicates that are defined by more than one clause require two choice points instead of one. Also, this initial instruction set contains fixed sequences of instructions: a new_subgoal_check_insert instruction is preceded by a try-type instruction and it is always followed by an allocate; similarly, a new_answer_check_insert instruction is followed by a deallocate instruction. We take care of these deficiencies by creating mega-instructions as follows:

- Tabled predicates defined by a single rule, are compiled using a tabletrysingle tabling instruction that includes the functionality of a try_me_else, new_subgoal_check_insert, and allocate sequence of instructions.

- To avoid unnecessary choice point creation, tabled predicates defined by more than one clause are compiled using the tabletry, tableretry, and tabletrust SLG-WAM instructions, rather than with the transformation presented in Section 4.5.1. Note that the tabled choice point created by the tabletry instruction is not disregarded until the completion of the subgoal. The try/retry/trust part of these instructions refers to the alternative clauses of the auxiliary Prolog predicate created by the transformation shown in the previous section.

- The failure continuation of the tabletrysingle and of the tabletrust instructions points to a completion instruction that is created *dynamically* and is responsible for:

[6] In general, if a clause contains n permanent variables, the extra environment cell is denoted as v_{n+1}. This extra permanent variable also explains the the first argument of the call instruction in Figure 12.

   – performing answer clause resolution (by dynamically triggering an appropriate number of answer_return instructions until the fixpoint condition is reached);

   – checking for the completion of the subgoal; and

   – deallocating the generator choice point set up by the tabletrysingle instruction and any frozen portions of the stack, if the subgoal is completed.

- Finally, the functionality of the new_answer_check_insert and deallocate sequence of instructions is folded into a single SLG-WAM instruction which we call new_answer.

Following these principles, the compiled SLG-WAM code for predicate p/2 in the program of Example 4.1.1 is shown in Figure 13. As mentioned, the completion and all the necessary answer_return instructions are generated and invoked dynamically.

L_1 :	tabletry	2	L_3	*TIP*	%
L_2 :	tabletrust	2	L_{10}		%
L_3 :	getpvar	v_1	r_2		% p(X,Z) :-
L_4 :	putpvar	v_2	r_2		% p(X,Y
L_5 :	call	3	p/2		%),
L_6 :	putpval	v_2	r_1		% p(Y,
L_7 :	putpval	v_1	r_2		% Z
L_8 :	call	3	p/2		%)
L_9 :	new_answer	2	v_3		% .
L_{10}:	getpvar	v_1	r_2		% p(X,Z) :-
L_{11}:	call	2	e/2		% e(X,Z),
L_{12}:	putpval	v_1	r_1		% q(Z
L_{13}:	call	2	q/1		%)
L_{14}:	new_answer	2	v_3		% .

Figure 13: *SLG-WAM code for predicate* p/2 *of Figure 3.*

The pseudocode for the tabletrysingle instruction is shown in Figure 14. The arguments of the subgoal are in the argument registers (the *Arity* of the subgoal is a parameter). The instruction first checks whether the subgoal S is new or already exists in the subgoal table. The subgoal table of the predicate is found by following the *TIP* (refer to Figure 9). We note that when tables are represented as tries, the subgoal check/insert operation can be done in one pass. If S is new, the instruction creates a table entry for it (a *subgoal frame*), pushes a *generator choice point frame* and a *completion stack frame* into the choice point and completion stacks respectively, and uses program clause resolution to obtain answers for the subgoal. If S already existed in the table, the instruction also checks whether S is *completed*; i.e. whether it is already known that all of its answers have been derived. If the subgoal is still incomplete, the computation is suspended by freezing the stacks and by setting up an *active subgoal frame* which is later going to be used to perform *answer resolution* for this

```
Instruction tabletrysingle(Arity, TIP)   /* Subgoal is in argument registers */
  SubgoalTable := get_subgoal_table(TIP);
  If (subgoal_check_insert(Subgoal,SubgoalTable) == new)
    Create and set up a subgoal frame SF for the Subgoal;
    Set up a generator choice point TCP to perform program clause resolution;
    Set the failure continuation of TCP to point to a completion instruction;
    Allocate a local environment;
    Set v_{n+1} (extra cell of local environment) to point to TCP;
    Push a new completion stack frame CSF onto the Completion Stack;
    Associate CSF with SF;                    /* see Section 4.6 */
    Branch to the next instruction to perform program clause resolution;
  else  /* Subgoal was not new to the evaluation */
    If (SF_Compl(Subgoal) == completed)
      Root := SF_AnsTrieRoot(Subgoal);    /* execute the answer trie code */
      Branch to Root to perform answer clause resolution;
    else    /* Subgoal was not new but it is still incomplete */
      Set up an active subgoal frame ASF for this subgoal instance;
      Set the failure continuation of ASF to point to an answer_return instruction;
      Freeze stacks and suspend the computation;
```

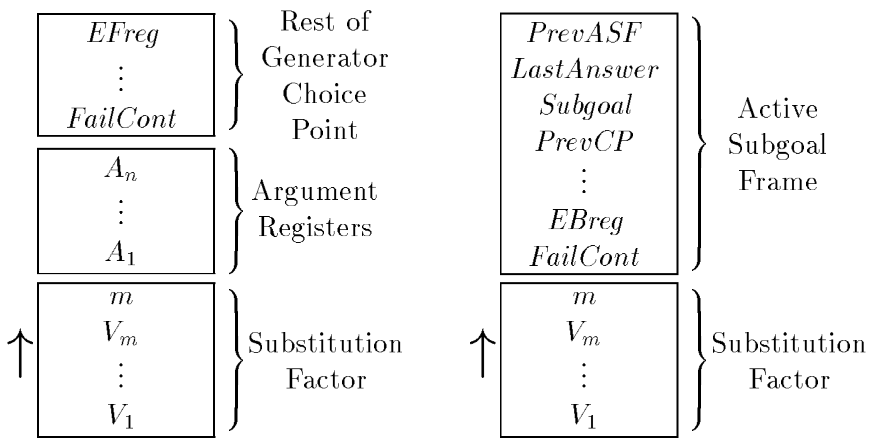
Figure 14: *The* tabletrysingle *instruction.*

subgoal instance. However, if the subgoal is completed, an optimization is possible: there is no need to freeze stacks or to create an active subgoal frame. Instead, following rule (iii) of the scheduling strategy of page 48, execution immediately branches to the root of its answer trie to immediately return its answers by backtracking through them. Doing so, information in a completed table can be used just as if it were represented by unit clauses.

The tabletrysingle (and tabletrust) instructions also place a completion instruction in the *FailCont* cell of the generator choice point. Recall that in the WAM the *FailCont* cell points to the instruction to be executed upon failure of the current clause; thus, the completion instruction will be executed after all program clause resolution has been performed for the subtree stemming from this generator choice point. We also note that the tabletry instruction is similar to the tabletrysingle instruction, but it also has a *Label* as an argument (cf. Figure 13) which is used to set the *FailCont* cell of the generator choice point to a tableretry or tabletrust rather than to a completion instruction. Finally, the tableretry and tabletrust instructions are analogous to their WAM counterparts retry and trust modulo handling tabled choice points rather than Prolog ones.

A note on substitution factoring *Substitution factoring*, an operation introduced in [50], exploits the property that all answers of a tabled subgoal G are subsumed by G.

Thus, each answer A of G can be represented as $G\eta_A$, where η_A is an *answer substitution* for G. The core idea of substitution factoring is to store only the answer substitution, and to create a mechanism of returning answers to active subgoals that is proportional to the size of η_A rather than the size of A. Operationally, this means that only substitutions for the variables $V_1, V_2, \ldots, V_m$ in G have to be stored in answer tables. In the SLG-WAM, substi-



(a) Generator Choice Point (b) Active Subgoal Frame

Figure 15: *Elements of the Choice Point Stack with Substitution Factoring.*

tution factoring is implemented as part of the subgoal_check_insert operation. This operation must fully traverse the subgoal to check if it is in the table or insert it if not. As it does so, it factors out dereferenced variable occurrences from the subgoal and places them in the choice point stack; this is done regardless of whether the subgoal was already present in the table or is newly encountered. We refer to this set of variables (and their substitutions) as the *substitution factor* of the subgoal. Since the subgoal_check_insert operation precedes the creation of either a generator choice point or an active subgoal frame (cf. Figure 14), the substitution factor is stored immediately *before* these frames (as shown in Figure 15). We also note that bindings to the variables of the substitution factor are trailed through forward execution, and are untrailed through backtracking just as bindings to argument registers are in the WAM. Further implementation issues and properties of substitution factoring can be found in [50].

When an answer to a tabled subgoal is detected, the dereferenced values of the substitution factor constitute *answer substitutions* which, if newly encountered, are copied from below the generator choice point into the appropriate answer table, and are directly returned to the generator instance of the subgoal. This is done via the new_answer SLG-WAM instruction which, for definite programs, is presented in Figure 16. This instruction is used as the last instruction of each clause of tabled predicates. As mentioned, new_answer requires a local environment cell to hold a pointer to the generator choice point (denoted as v_{n+1} in Figure 16). Using v_{n+1}, the instruction accesses the generator choice point of the subgoal which,

```
Instruction new_answer(Arity, v_{n+1})
    answer_table := SF_AnsTrieRoot(get_subgoal_from_pvar(v_{n+1}));
    η_A := locate_substitution_factor(Arity, v_{n+1});
    if (answer_check_insert(η_A, answer_table) = new)
        /* the answer substitution was inserted */
        deallocate local environment;
        set the program pointer P to the continuation pointer CP;
    else
        fail;   /* η_A was already present in the answer table */
```

Figure 16: *The* new_answer *instruction (for Definite Programs).*

in combination with the *Arity* of the subgoal, gives it access to the substitution factor, and to the *subgoal frame* which, in turn, contains information about the subgoal's *answer table* and the *answer return list* (see Figures 15, 8, and 10). Then, the operation of new_answer is to perform a check to determine whether the answer substitution η_A has already been entered into the answer table. If it has, the derivation path fails, a vital step for ensuring termination. If η_A is new, new_answer will deallocate the environment and set the WAM program register to the continuation pointer (**CP** register), to continue with forward execution. This operation, in effect, returns this new answer to the *generator* instance of the subgoal (the node that initiated this computation). As a result, the SLG-WAM executes *first-call optimization*, i.e. it optimizes away the first occurrence of each tabled subgoal; *generators* do not need to be suspended and stacks do not need to be frozen for single occurrences of tabled subgoals.

Answers of tabled predicates are scheduled for return to the applicable *active* subgoals through an answer_return SLG-WAM instruction whose pseudocode is presented in Figure 17. Recall that active subgoals are suspended and that their frames, whose failure continuation is set to point to an answer_return instruction (see Figure 14), are maintained in a chain. Using this chain, answers are returned to one active subgoal at a time until all currently derived answers of the answer return list have been consumed by all active instances of a given subgoal. Frames of active subgoals constitute environments of suspended computations. As explained in Section 4.2, to return answers to these suspended environments, a *resume* operation should first take place. In other words, the variable bindings of that node in the search tree have to be restored using a restore_bindings operation; an operation which can be costly. Thus, by returning all unconsumed answers to the same active subgoal, rather than one answer to all active subgoals, the amount of environment switching performed by the SLG-WAM is minimized. As mentioned, the scheduling of the answer_return instructions is performed dynamically. This is done as part of the completion instruction which is described in Section 4.6.

```
Instruction answer_return
 ASF := breg;           /* current breg points to the active subgoal frame */
 restore_bindings(old_breg,ASF);   /* restore environment of active subgoal */
 if (the last answer consumed by this ASF
     is not the last answer of the answer return list)
   /* let answer be the first unconsumed answer of the answer return list */
   ASF_LastAnswer(ASF) := answer;    /* mark answer as consumed */
   load answer from the answer trie into the substitution factor below the ASF;
   set the program pointer P to the continuation pointer CP;
 else /* initiate return of answers to another (the previous) active subgoal */
   breg := ASF_PrevASF(ASF);
   fail;
```

Figure 17: *The* answer_return *instruction (for Definite Programs).*

4.5.3 Dynamic Compilation of Answer Tries

In this section we describe how answer substitution tries can be represented as a set of WAM instructions. One of the advantages of using this representation is that when a completed tabled subgoal is encountered, its answers trie, instead of being interpreted, is simply executed as WAM code. A direct result (of this WAM code execution), is that the answer substitutions of the subgoal are constructed on the heap and are returned one at a time through backtracking.

To motivate the new WAM instructions introduced for this purpose, we first show how an (answer substitution) trie can be represented as regular Prolog clauses. We then consider the (mostly) standard WAM code segments to implement those clauses, and finally we create "mega"-instructions that are our new WAM instructions which constitute a space efficient representation of answer substitution tries. We do this development through the following example. Assuming that *substitution factoring* is employed, consider the answer substitution trie of a subgoal that contains three variables shown in Figure 18. The four answers[7] in this trie could be represented simply by the Prolog facts of Figure 19(a). Now, note that an alternative representation of these facts could be the one where the facts are prefix-factored into a *full-trie* [17] represented as a set of Prolog clauses (shown in Figure 19(b)). In the same figure, note that each clause corresponds to a single edge in the answer substitution trie of Figure 18, and that the order of the clauses reflects a breadth-first left-to-right traversal of the edges of the trie.

Throughout this section we assume that when an answer trie of a completed subgoal containing a set of n variables is about to be called to backtrack through its answers, the

[7]For simplicity of presentation we consider linear answers first, and describe how non-linearity is handled later.

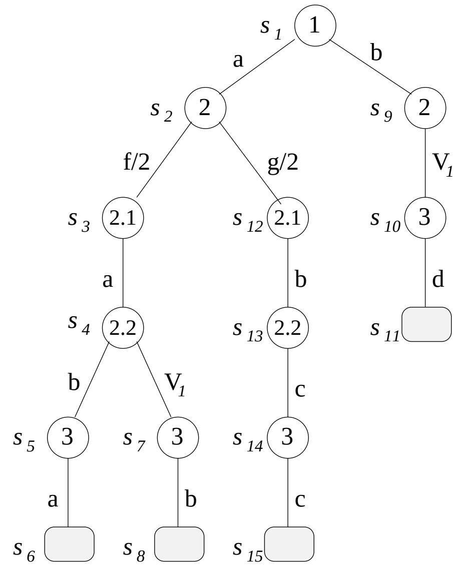


Figure 18: *Answer Trie.*

first n registers have been initialized to hold the variables that are to be bound to the answers to be returned. This initialization can be easily performed while traversing the subgoal in the subgoal trie.

Figure 20 shows WAM-like code segments for the first three clauses of Figure 19(b). The code for the first two clauses starts with a choice-point instruction, here **try_me_else** and **trust_me_else** instructions. The second instruction is a **get_*type*** instruction. The third instruction is new to the WAM, a **shift_left** instruction. Its function is to shift all the registers

(a)

```
rt(a,f(a,b),a).
rt(a,f(a,V),b).
rt(a,g(b,c),c).
rt(b,V,d).
```

(b)

rt(a,X_2,X_3) :- rt_a(X_2,X_3).
rt(b,X_2,X_3) :- rt_b(X_2,X_3).

rt_a(f($X_{1.1}$,$X_{1.2}$),X_2) :- rt_af($X_{1.1}$,$X_{1.2}$,X_2).
rt_a(g($X_{1.1}$,$X_{1.2}$),X_2) :- rt_ag($X_{1.1}$,$X_{1.2}$,X_2).

rt_b(V,X_2) :- rt_bV(X_2).

rt_af(a,X_2,X_3) :- rt_afa(X_2,X_3).

rt_ag(b,X_2,X_3) :- rt_agb(Y,X_3).

rt_bV(d).

rt_afa(b,X_2) :- rt_afab(X_2).
rt_afa(V,X_2) :- rt_afaV(X_2).

rt_agb(c,X_2) :- rt_agbc(X_2).

rt_afab(a).

rt_afaV(b).

rt_agbc(c).

Figure 19: *Two possible representations of the answer trie of Figure 18.*

rt_1:	try_me_else	rt_2	rt_2:	trust_me_else	fail	rt_a_1:	try_me_else	rt_a_2
	get_constant	a, A_1		get_constant	b, A_1		get_structure	f/2, A_1
	shift_left	1		shift_left	1		shift_right	1
	execute	rt_a/2		execute	rt_b/2		unify_variable	V_1
							unify_variable	V_2
							execute	rt_af/3

Figure 20: *WAM-like instructions for the first 3 clauses of Figure 19(b).*

to the left by some number of positions (here one). This is needed to set up the arguments for the final instruction, the **execute**, which branches to the next clause. Now consider the code for the third clause, whose first argument is a structure. Here again the first instruction is a choice-point instruction, and the second instruction is a **get_*type*** instruction, this time for structures. Then comes a new instruction, a **shift_right** instruction whose purpose is to shift the registers right enough positions to make room for the arguments of the structure symbol.[8] This instruction is followed by a series of argument-construction instructions, **unify_variable**, one for each argument. Finally, there is again an **execute** instruction to branch to the next clause.

Code segments like the ones in Figure 20, when executed, will construct the answers one-at-a-time on the stack through backtracking, as desired. The idea now is to create single WAM instructions that perform the work of each of these code segments (and the others that could arise). We use the term *trie instructions* to refer to these new instructions. Note that in each case, besides the use of possibly different *choice* and **get_*type*** WAM instructions, there are essentially three major parameters to the code segment: (1) the constant, structure symbol, or variable to match ($Symbol$),[9] (2) the address of the next code segment down the trie ($ContLabel$) and, (3) the address of the alternative to try on failure ($FailLabel$). So, the general form of the trie instructions should be the following:

$$\textsf{trie\_}\textit{choice}\textsf{\_}\textit{type} \quad Symbol, \quad ContLabel, \quad FailLabel$$

In analogy to the WAM, since each of the trie instructions may appear as the only, the first, an intermediate, or the last instruction in a sequence of alternatives, the different *choice* possibilities are **do**, **try**, **retry**, and **trust**, respectively. As for *types*, constants, structures, lists (specialization of structures), uninstantiated and instantiated variables should be handled. Figure 21 presents the names of the trie instructions. In order to handle answer tries of subgoals that have no variables, another trie instruction needs to be introduced, named **trie_proceed** which has exactly the functionality of the WAM's **proceed**. Note that this instruction also needs to be executed after the instruction corresponding to an edge of the answer trie leading to a leaf node.

As a concrete example of how the trie instructions are used, Figure 22 shows the WAM code generated for the clauses of Figure 19(b), and, as a consequence, for the answer trie of

[8]The required number of positions is always one less than the arity of the structure symbol.

[9]Note that the argument register that is involved in the **get_*type*** instructions is always register 1 (A_1).

	Unique	*First*	*Intermediate*	*Last*
Constant	trie_do_constant	trie_try_constant	trie_retry_constant	trie_trust_constant
Structure	trie_do_structure	trie_try_structure	trie_retry_structure	trie_trust_structure
List	trie_do_list	trie_try_list	trie_retry_list	trie_trust_list
Variable	trie_do_variable	trie_try_variable	trie_retry_variable	trie_trust_variable
Variable	trie_do_value	trie_try_value	trie_retry_value	trie_trust_value

Figure 21: *Trie instructions for all possible* choice *and* type *combinations.*

Figure 18. Since no choice points are laid down for trie_do_? instructions, their $FailLabel$ is not used. Observe the correspondence between the labels of the instructions in Figure 22 and the names of the atoms in the clauses of Figure 19(b). For all clauses that are facts, the label *proceed* was used as the $ContLabel$ of the corresponding trie instruction to indicate that the final operation of the trie instruction should be a proceed rather than an execute. A slight optimization is to create specialized versions of trie instructions that know in advance that their $ContLabel$ is a *proceed*. Note that such specialized instructions are needed only for constants and variables.

rt_1:	trie_try_constant	a,	rt_a_1,	rt_2
rt_2:	trie_trust_constant	b,	rt_b,	fail
rt_a_1:	trie_try_structure	f/2,	rt_af,	rt_a_2
rt_a_2:	trie_trust_structure	g/2,	rt_ag,	fail
rt_b:	trie_do_variable	V,	rt_bV	
rt_af:	trie_do_constant	a,	rt_afa_1	
rt_ag:	trie_do_constant	b,	rt_agb	
rt_bV:	trie_do_constant	d,	*proceed*	
rt_afa_1:	trie_try_constant	b,	rt_afab,	rt_afa_2
rt_afa_2:	trie_trust_variable	V,	rt_afaV,	fail
rt_agb:	trie_do_constant	c,	rt_agbc	
rt_afab:	trie_do_constant	a,	*proceed*	
rt_afaV:	trie_do_constant	b,	*proceed*	
rt_agbc:	trie_do_constant	c,	*proceed*	

Figure 22: *WAM code for the clauses of Figure 19(b).*

The operational semantics the trie WAM instructions should be easily deducible if one thinks of them as macros that define WAM code segments like those of Figure 20. For example, the three code segments of Figure 20 present the operations performed (for some values of the parameters) by the trie_try_constant, trie_trust_constant, and trie_try_structure instructions, respectively. We note that the shift_left and shift_right operations are new to the WAM, but they could be implemented efficiently in a logic programming system that stores the registers as an array simply by modifying the base of that array. Alternatively, a separate array of pseudo-registers could be used for the trie instructions only, which would allow it to perform efficiently as a *register stack*. The latter is the implementation scheme chosen by XSB. Non-linearity is handled by adding another array to the SLG-WAM, called the *var-array*. The trie_?_variable instructions initialize the indicated var-array entry to be a free variable, allocated on the heap, setting the top element of the register stack to point to it. The trie_?_value instructions unify the top register of the register stack with the indicated var-array variable.

Observe that in a tabling framework that performs variant checks for subgoals, the technique of substitution factoring allows all uses of the get_*type* subinstructions in the answer trie to actually *build* their values, i.e., to run in write mode. We also note that we have not discussed indexing in our code representation of tries. Indeed, hash-based indexing is used to perform the check/insert operation efficiently when constructing the tries, and the set of trie instructions described here is extended with two more instructions to account for those hashes. However, since the execution of these instructions is used to backtrack through all answers, the hashes do not provide any extra efficiency once a subgoal is completed. In fact, they slightly complicate the dynamic compilation, and impose a small but unnecessary overhead to the execution.

Finally, observe that the instructions that allow dynamic compilation of tries are immediately applicable in the general context of efficiently creating and accessing dynamically created facts. In particular, they can be viewed as an alternative instruction set that can be used for the compilation of asserted facts. Naturally, the full unification power of the *get*_type instructions would be needed in such a case. In [50], besides performance results that show the superiority of *compiled answer tries* versus interpreted data structures, we also provide experimental demonstration of the performance gains of using this instruction set to access and create asserted facts, over extant compilation methods for dynamic code.

We end this section by stating a relatively straightforward but still important property of representing tries as WAM code. This property is that all common prefixes of the terms in a trie are shared during execution of the WAM code.

Proposition 4.5.1 ([50]) *When backtracking through the terms of a trie that is represented using the trie WAM instructions, each edge of the trie is traversed only once.*

4.6 Completion in Definite Programs

With the scheduling strategy we employ, the completion instruction for definite programs is responsible for performing the following two, conceptually distinct, actions:

1. Ensuring that all answers in the tables have been returned to all instances of tabled subgoals by scheduling appropriate answer_return instructions.

2. Determining (mutually dependent) sets of subgoals that have exhausted all possible answer and program clause resolution and reclaiming the stack space occupied by these subgoals.

Notice the correspondence of these actions with the answer_return_loop and the completion_check instructions of Figure 13. The functionality of the first action, which we also refer to as *pre-completion*, is self-explanatory and its implementation is sketched in Figure 23. Further details on pre-completion can be found in [61, 27]; we only note that the need for pre-completion generally depends on the scheduling strategy that is employed; in particular, there are scheduling strategies that do not require a pre-completion step (however these strategies require a more fine-grained instruction set for the ANSWER RETURN SLG operation). Assuming that pre-completion can be appropriately initiated, we focus on the second action of the completion instruction for definite programs which is more subtle.

```
Procedure schedule_answers(subgoal)
  if (there are answers of subgoal not returned to all active subgoals)
    Link all active subgoal frames (with unconsumed answers) into the breg chain;
    fail;    /* to execute answer_return instructions */
             /* by picking up the failure continuation */
```

Figure 23: *Sketch of the pseudocode that implements pre-completion.*

Strictly, the second action, which reflects the functionality of the SLG COMPLETION operation, is not required for the correct evaluation of definite programs. The evaluation of a query against a program could end with pre-completion; i.e. the evaluation could terminate when the fixpoint has been reached; when all generated answers have been returned to all instances of tabled subgoals. Nevertheless, *incremental completion*, an operation first introduced in [12], can significantly reduce the space requirements of SLG resolution by reclaiming the stack space occupied by subgoals that are detected to be completely evaluated. For example, note how incremental completion affects the choice point stack of Example 4.1.1 shown in Figure 4, by incrementally unfreezing and reclaiming the stack space for subgoals `p(c,Z)`, `p(b,Z)`, and finally `p(a,Z)`. Furthermore, completion of subgoals enables the *completed table optimization* described as part of the tabletrysingle instruction. The later, besides avoiding unnecessary freezing of the stacks, allows backtracking through answer clauses in completed tables to be performed at speeds comparable to backtracking through compiled WAM code (see [50]).

To efficiently perform incremental completion, the SLG-WAM contains an area of memory new to the WAM, the *Completion Stack*. Because all operations are scheduled on the choice point stack, when pre-completion has been performed and the engine fails over a generator choice point (the choice point associated with the first instance of each tabled subgoal), that subgoal has been completely evaluated. What this means is that as long as the subgoal is not part of some larger component, it will have exhausted all program and answer clause resolution. The purpose of the completion stack, which conceptually can be viewed either as an extension of the choice point stack or as a restriction of the choice point stack to just choice point frames of generators, is to keep track of the subgoals in each component. In general, to do so, precise information about dependencies between subgoals should be explicitly maintained. Since the only reason for incremental completion in definite programs is efficiency, a design decision of the SLG-WAM for definite programs was to keep only an easily maintainable approximation of the subgoal dependency graph, rather than exact information about it. This approximation can be extended to lazily construct the exact dependency graph, whenever this is needed, as will be presented in the following chapter.

To perform approximate incremental completion the SLG-WAM maintains, for each subgoal A, the deepest subgoal B in the completion stack which A or any subgoal on top of A *may* depend upon. When A and all subgoals on top of A have exhausted all program and answer clause resolution, A is checked for completion. If the subgoal B is deeper in the completion stack than A, A may depend upon subgoals that appear below it, and therefore cannot be completed. Otherwise, A and all subgoals on top of A are completely evaluated. As an aside, we note that for the program of Figure 3, each tabled subgoal can be completed after pre-completion and failure over its generator choice point since each component consists of a singleton set of subgoals, but this situation is not the case in general.

In terms of implementation, a *completion stack frame* is pushed onto the completion stack when the generator choice point is created (by executing a tabletrysingle or a tabletry instruction for this subgoal — see Figure 14), and is popped off when that subgoal is determined as completely evaluated by execution of a completion instruction. Thus, the following is an invariant of the abstract machine for the tabled evaluation of definite programs:

Invariant I *The completion stack contains a completion stack frame for a subgoal S only if S is in the system but is not yet completed by applying the* COMPLETION *operation.*

For definite programs, the format of the completion stack frame can be thought as follows:

Subgoal	*DFN*	*DirLink*	*MinLink*

In the above data structure, *DFN* is a unique depth-first number (assigned through a global counter), *DirLink* keeps track of the deepest direct subgoal dependency, while the motivation behind *MinLink* is to keep an (over) estimate of the subgoal's indirect dependencies by propagating minimum *DirLink* values backwards as goals are checked for completion. We briefly present how fields of the completion stack frames are updated:

- When the generator instance of a subgoal A is encountered by the evaluation, a unique depth-first number is assigned to A, a new frame is pushed onto the completion stack, and $DirLink(A)$ and $MinLink(A)$ are initialized to $DFN(A)$.

- When an active instance of a tabled subgoal B is encountered, if B is not completed, and B's nearest tabled ancestor in the search tree is A, then set

$$DirLink(A) := min(DirLink(A), DirLink(B)). \quad (1)$$

- When checking for completion of A, A may not be completely evaluated and as a result A cannot be completed if $DirLink(A) \neq DFN(A)$, or $MinLink(A) \neq DFN(A)$. Let B be the subgoal that is immediately below A on the Completion Stack. Set

$$MinLink(B) := min(MinLink(A), DirLink(A)). \quad (2)$$

Based on these rules and the format of the completion stack frame, the following notion can be defined:

Definition 4.6.1 (*Leader*) In an SLG-WAM evaluation of definite programs a subgoal S is called a *leader* iff the completion stack frame associated with S satisfies the following condition:

$$DFN(S) = DirLink(S) = MinLink(S)$$

Completion stack frames of subgoals $\{S_1, S_2, \ldots, S_n\}$ that are leaders partition the completion stack into components $\{AC_1, AC_2, \ldots, AC_n\}$ with the property that no subgoal of component AC_i depends on a subgoal of component AC_j, where $i > j$. As a result, the topmost leader S_n can be used to determine when the set of subgoals in the topmost component AC_n of the completion stack can be completed. This property is the basic idea behind incremental completion. As for terminology, we sometimes call these components *approximate strongly connected components* (*ASCCs*), and we refer to leaders as *leaders of an ASCC*. The SLG-WAM completion instruction for definite programs is presented in Figure 24.

```
Instruction completion
subgoal := TCP_subgoal(breg);
schedule_answers(subgoal);              /* pre-completion part */
if (subgoal is a leader of an ASCC, A) /* according to Definition 4.6.1 */
    Mark as completed all subgoals in A;
    Reclaim the stack space of subgoals in A and adjust the freeze registers;
else breg := TCP_PrevCP(breg);
fail;
```

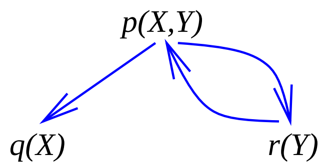
Figure 24: *The* completion *instruction (for Definite Programs).*

Contrary to definite programs where completion is used for efficiency, for programs containing negation the COMPLETION operation is essential for their correct evaluation. In general, the truth value of a negative literal can be determined only after the completion of its positive counterpart. For example, $\neg S$ is true if and only if subgoal S finitely *fails*; i.e. S is completely evaluated and has no answers. Also, as discussed in the next sections, for the class of left-to-right dynamically stratified programs COMPLETION is necessary in order to decide when to unsuspend clauses containing subgoals whose selected literal is negative. As a result, the algorithm used for the incremental completion of definite programs is insufficient for the evaluation of LRD-stratified programs. In the presence of negation the information about subgoal dependencies has to be exact; completing all subgoals of an approximate strongly connected component is not a correct strategy in general.

Using a stack to approximate a subgoal dependency graph can lead to over-approximation of SCCs, as shown by the following example in which $q(X)$ is not detected to be completely evaluated as soon as it can be.

Example 4.6.1 ([12]) Given the following program, Figure 25 depicts the evaluation of the query `?- p(X,Y).` along with the subgoal dependency graph and completion stack at the time of completion of $p(X, Y)$. The order of entries in the Completion Stack reflects the *DFN* of the subgoals. Subgoal $q(X)$, with *DFN* 1 is trapped below $r(Y)$ with *DFN* 2,

```
:- table p/2, r/1, q/1.

p(X,Y) :- q(X), r(Y).
p(c,a).

q(a).
q(b).

r(c).
r(X) :- p(X,Y).
```



Subgoal	DFN	DirLink	MinLink
r(Y)	2	0	2
q(X)	1	1	0
p(X,Y)	0	0	0

Figure 25: *A Trapped Component (consisting of a single subgoal).*

because its *MinLink* slot inherits the *DirLink* value from subgoal $r(Y)$. As a result, $p(X, Y)$ is the only leader, all three subgoals end up in the same approximate SCC and are going to be completed simultaneously. ■

Notice, however, that for space reclamation purposes through unfreezing stacks, detection of completion of $q(X)$ is not useful. To see this, first note that there is a one-to-one correspondence between completion stack frames and generator choice point frames; and that their order is the same. Thus if $r(Y)$, which is above $q(X)$, depends on $p(X, Y)$ below it, there must be an active node with selected literal $p(X, Y)$ above $q(X)$ in the choice point stack. As a result, all WAM stacks remain frozen by the active node regardless of whether $q(X)$ is completed. Space frozen by $q(X)$ therefore cannot be unfrozen until the leader of the SCC, $p(X, Y)$, is completed, so that the use of *ASCCs* is suitable for space reclamation in definite programs. If a program contains negation, however, exact SCC detection is needed for its correct evaluation. Given exact SCC detection, a large class of stratified programs can be evaluated by suspending computations that depend on a negative literal, and resuming them only after the subgoal is completed. How this can be done in the SLG-WAM framework is the subject of the following chapter.

Chapter 5

The Abstract Machine for LRD-Stratified Programs

5.1 Tabling for LRD-Stratified Programs

Determining when all possible answers have been derived for a set of subgoals — that is when these subgoals have been *completely evaluated* — is a central issue in designing a tabling engine for LRD-stratified programs. To correctly determine this, and hence to implement the COMPLETION operation when negation may be present in programs requires exact information about subgoal dependencies. To represent dependencies, we use the notion of a *Subgoal Dependency Graph* (*SDG*) of an SLG system. The *SDG* is a directed graph that characterizes the run-time dependencies between non-completed subgoals in the SLG system as these as set up during the evaluation of a query against a program. Given a subgoal on the head of a rule, there are directed arcs in the subgoal dependency graph to each subgoal set up during the evaluation of the body of that rule. Assuming, as we do throughout this document, a complete left-to-right literal selection strategy, these intuitive definitions are formalized as follows.

Definition 5.1.1 (*Direct Subgoal Dependency*) Let P be a given program. We say that a tabled subgoal $p(\overline{t})$ *directly depends on* tabled subgoal $q_i(\overline{b_i})$ iff there is an SLG clause instance:

$$p(\overline{a}) :- l_1(\overline{b_1}), \ldots, l_i(\overline{b_i}), \ldots, l_n(\overline{b_n}).$$

such that $p(\overline{a})$ is an instance of $p(\overline{t})$, and where each $l_k(\overline{b_k})$ is either a positive or negative literal having $q_k(\overline{b_k})$ as its subgoal, and at least one ground instance of each literal $l_j(\overline{b_j}), 1 \leq j \leq i-1$ is true in the model of P. If $l_i(\overline{b_i})$ is a positive (negative) literal, then we say that subgoal $p(\overline{t})$ *directly depends positively* (*negatively*) *on* subgoal $q_i(\overline{b_i})$.

Definition 5.1.2 (*Subgoal Dependency Graph*) Let P be a given program, Q be a query to this program, and let $\mathcal{S}$ be an SLG system. The *Subgoal Dependency Graph* $SDG(\mathcal{S}) = (V, E)$ of $\mathcal{S}$ (or equivalently of the program-query pair $\langle P, Q\rangle$) is a directed graph in which V is the set of non-completed subgoals in $\mathcal{S}$ and $(s_i, s_j) \in E$ iff subgoal s_i directly depends on subgoal s_j. Edge (s_i, s_j) is labeled *positive* if s_i directly depends positively on s_j, otherwise it is labeled *negative*. □

Besides the five operations for definite programs presented in Section 4.1, for LRD-stratified programs we need operations for handling clauses whose selected literal is negative. These are FLOUNDERING, NEGATION FAILURE, NEGATION SUCCESS, and NEGATION SUSPENSION. In the following, let C be the clause containing the selected literal L $(= \neg B)$.

FLOUNDERING If L is a non-ground negative literal, then the evaluation is floundered.

NEGATION FAILURE If L is a ground negative literal $\neg B$, and B currently has an answer, then remove the clause C from the system.

NEGATION SUCCESS If L is a ground negative literal $\neg B$, where B has no answers and is marked as completely evaluated, then remove this literal from the clause C.

NEGATION SUSPENSION If L is a ground negative literal $\neg B$, and B is not marked as completed, then change the status of C to *suspended*.

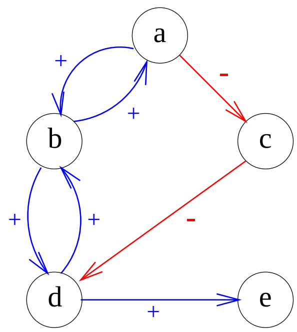
The first three operations for handling negative literals should be self-explanatory. For NEGATION SUSPENSION, recall that efficient evaluation of programs containing uses of fixed-order stratified negation requires an important point of departure from the evaluation technique previously described in [14]. This technique breaks the fixed order of computation for C (by applying the SLG DELAYING transformation) if L is a ground negative literal whose subgoal (i) has no answers, and (ii) is not completely evaluated. After delaying L, the evaluation may resolve further literals to the right of L potentially expanding the search space of the program. An alternative approach is to *suspend* the selected literal L, and, as a result, the clause C in which this literal appears. In such a case, we say that C is *suspended on* (the completion of) B, and that subgoal B has *negation suspensions* on it. Note that the situation in which the subgoal B is not in the system vacuously satisfies the conditions for the suspension of literal $\neg B$. In such a case, a *generator* node is created for the subgoal, and program clause resolution is used to derive answers for it. The clause C containing $\neg B$ remains *suspended* until one of the conditions that caused its suspension ceases to hold. That is, C remains suspended until an answer is derived for B (at which time its status will change to *disposed*), or until B is *completed* by invoking a SLG COMPLETION operation (in which case its status will return to *active*).

So far, no mention has been made of how to determine when a set of subgoals has been completely evaluated. Example 5.1.1 illustrates some of the difficulties involved.

Example 5.1.1 Let P be the LRD-stratified program in Figure 26(a) for which the query `?- a.` is to be evaluated. The execution of this query against P causes cascading suspensions, of `a` on `c` and `c` on `d`, as seen from the SDG shown in Figure 26(b). The only subgoal

```
:- table a/0,b/0,c/0,d/0.

a :- b, ¬ c.

b :- a.
b :- d.
b.

c :- ¬ d.

d :- b, e.
```

(a) A LRD-stratified Program



(b) The SDG for the query `?- a`.

Figure 26: *Program showing the need for early completion.*

that has answers is `b`, and the only subgoal that could be completed is `e`. Note that without disregarding the dependencies of subgoal `b` (i.e. removing subgoal `b` from the subgoal dependency graph), the program appears to contain a loop through negation (subgoals `a`, `b`, `c`, and `d` form a strongly connected component). Observe that the dependencies of subgoal `b` are irrelevant from the point of view of the well-founded semantics; they cannot contribute in any way to `b`'s truth value as the latter is already determined in the interpretation defined by the current SLG system (see Section 3.3.6). Also notice that simply abolishing negation suspensions of subgoals that have answers does not break the negative loop (all dependencies to and from `b` are positive). ■

Before addressing how to properly detect the completion of subgoals in LRD-stratified programs, we first connect the notion of a subgoal dependency graph of an SLG system with the notion of a completely evaluated set of subgoals (which was also presented as Definition 3.3.9).

Definition 5.1.3 (*Completely Evaluated*) Given a particular state $\mathcal{S}$ of an SLG evaluation, i.e. an SLG system, a set Σ of subgoals is completely evaluated if either of the following conditions is satisfied:

1. Σ consists of a single subgoal A having an answer that is a variant of A (i.e. A succeeds);

2. Σ is a strongly connected component (SCC) of $SDG(\mathcal{S})$ that depends on no other SCC (termed an *independent SCC*), no clause of subgoals in Σ is *suspended*, and all applicable SLG operations have been performed for each subgoal instance in Σ. □

The first condition will be referred to as *early completion* of subgoals. Note that early completion is *necessary* to evaluate the program in Example 5.1.1 using a fixed left-to-right computation rule. Besides its necessity for the evaluation of LRD-stratified programs, early completion is an operation which is applicable, and in fact can be of benefit, even when programs do not contain negation (as will be demonstrated in Chapter 7).

The following theorem, which is essentially a restated version of Theorem 3.4.3 from Chapter 3, indicates the validity of the approach outlined.

Theorem 5.1.1 *The operations* NEW SUBGOAL*,* NEW ANSWER*,* ANSWER RETURN*,* COMPLETION*,* NEGATION SUSPENSION*,* NEGATION FAILURE*, and* NEGATION SUCCESS *suffice for the evaluation of non-floundering queries to ground LRD-stratified programs.*

5.1.1 LRD-Stratified SLG-WAM: Overview

In order to evaluate LRD-stratified programs, five main changes are made to the abstract machine for definite programs that was described in the previous chapter: addition of a mechanism for early completion that eliminates irrelevant dependencies between subgoals, a mechanism for explicit maintenance of subgoal dependencies, suspend and resume operations for negative literals, implementation of a stratified negation operation, and a new completion instruction with the ability to perform the COMPLETION operation based on exact subgoal dependencies. We discuss each of these additions in turn.

5.2 On the Early Completion of Subgoals

The previous section motivated the need for early completion if LRD-stratified programs are to be evaluated without the use of the DELAYING transformation of SLG resolution. In this section we present the early completion operation from a slightly different perspective, we discuss its implementation in the framework of the SLG-WAM, and we conclude by briefly discussing some of its properties.

Recall that whenever an answer is derived for a particular subgoal, a check is performed to determine whether this answer already exists in the answer table associated with the subgoal. If it does, the derivation path fails. Consider now what happens in the case where a ground subgoal S has an answer A in its answer table. Notice that A will be identical to the subgoal S. Furthermore, note that after A is inserted in the answer table of S, no other answer will be inserted. Any answer that might be derived for S would have to be identical to A, i.e. it would already exist in the answer table, consequently resulting in a failing derivation path. Thus, a ground subgoal S can be considered completely evaluated upon the addition of its first answer into its answer table.

Extending the above argument, consider the case of a (possibly non-ground) subgoal S that gets an answer A which is identical up to variable renaming to S. Note that A is the most general answer of S. Any other answer A' of S will be subsumed by A, and thus will be logically implied by an answer that is already in the answer table of S. Consequently, upon the derivation of A, subgoal S can be considered as being completely evaluated: from a logical point of view all answers for S have already been derived, as any new answer will be subsumed by an existing one. As both program and answer clause resolution *will not* *produce any new information for S*, their further continuation is unnecessary, and therefore could be pruned.

These observations suggest the following operation which we will call *early completion* of subgoals. Upon the derivation of an answer which is identical up to variable renaming to its associated subgoal S, S is marked as completely evaluated, and the operation that performs the SLG COMPLETION transformation for a set of subgoals is scheduled for immediate execution with $\{S\}$ as the set of subgoals. Finally, execution of the unselected alternative clauses of subgoal S is bypassed, thus avoiding all program and answer clause resolution that has not yet been initiated by the execution of S. Note however, that early completion does not prune computations that have already been initiated. We illustrate the effect of early completion with the following example.

Example 5.2.1 Consider the execution of the query $Q =$ `p(a,Y)` with respect to the program in Figure 27(a). Notice that predicate `p/2` does not terminate under SLD evaluation; it does so under SLG. Without early completion, the set of tabled subgoals generated during the

```
:- table p/2.

p(X,Y) :- q(X), p(X,X).
p(X,Y) :- r(X).
p(X,Y) :- p(b,Y).

q(a).     r(a).
q(b).     r(b).
q(c).
```

(a)

Without early completion	
Subgoals	`p(a,Y), p(a,a), p(b,Y), p(b,a), p(b,b)`
Answers of `p(a,Y)`	`p(a,Y), p(a,a), p(a,b)`

With early completion	
Subgoals	`p(a,Y), p(a,a)`
Answers of `p(a,Y)`	`p(a,Y)` [`p(a,a)`]

(b)

Figure 27: *Effect of Early Completion.*

execution of Q, as well as set of answers of subgoal `p(a,Y)` is shown in the first table of Figure 27(b). When early completion is employed, the third clause of `p/2` is *not* examined and the set of subgoals that are generated is shown in the second table of Figure 27(b). Both of these subgoals are early completed when answers for them are derived from the second clause of `p/2`. We finally note that whether the answer `p(a,a)` will be added to the answer table of subgoal `p(a,Y)` is up to the strategy that the `new_answer` instruction is using regarding the issue of adding answers to answer tables of subgoals that are already marked as completely evaluated; an issue that is beyond the scope of this document. ■

As illustrated by the above example, early completion makes the computation demand-driven, allowing examination of only the part of the program that is necessary for answering queries when clauses are selected in textual order; i.e. when following Prolog's clause selection strategy. As a result, it can significantly reduce both the space and the time needed for the execution of queries. Note however, that early completion does not affect the execution of all queries. As an extreme example, clearly early completion does not affect the execution of subgoals that finitely fail. As a non-trivial example, early completion would not affect the set of subgoals generated as a result of executing the query $Q' = \mathtt{p}(\mathtt{X}, \mathtt{c})$ against the program in Figure 27(a).

5.2.1 Implementation of Early Completion

In the SLG-WAM framework, early completion is implemented as part of the new_answer instruction. Recall from Section 4.5 that this instruction is laid down as the last instruction of each clause of tabled predicates and that it uses the answer substitution bindings from the generator choice point in the answer check/insert step. To perform early completion the engine must check if the answer is a variant of the call; i.e. whether all of the answer substitution variables is are still single occurrence variables. Note that this condition trivially includes the case of deriving an answer for a ground subgoal. In such a case, the subgoal frame, which is accessible through the generator choice point, is marked as completed, and a completion instruction is placed in the *FailCont* cell of the generator choice point (see Figure 8 in Section 4.3.1), thus bypassing any remaining tableretry and tabletrust instructions for that subgoal.

Note that as a consequence of the early completion operation, subgoals can be determined to be completely evaluated possibly before the execution of a completion instruction. In other words, early completion violates the assumption of the strictly stack-based completion algorithm employed in definite programs. This issue is further discussed in Section 5.6.

5.3 Maintenance of the Subgoal Dependency Graph

Recall that in the SLG-WAM, either a generator choice point or an active subgoal frame is created for each occurrence of a tabled subgoal that is not yet completed. Thus, vertices of the *SDG* are present in the SLG-WAM, and the only missing piece of information for having the SDG explicitly maintained is from each choice point of a tabled subgoal, knowledge about the choice points of its children in the search tree. As usual in the WAM, dependency information corresponding to *SDG* edges is more easily maintainable from body to head than from head to body as specified in Section 5.1. Thus, rather than maintaining the SDG, its transpose (denoted as SDG^T) is maintained.

In the WAM, information about the state of the computation is recorded in global registers. We therefore introduce another global register, which we will call **PT** *register* (short for *parent tabled register*). All tabled or non-tabled choice point frames used for program or answer clause resolution are expanded with an extra field (called *PTreg*) that records the value of **PT** register at the time of choice point creation. This register is updated so that it always points to the generator choice point frame of the tabled subgoal that is the nearest ancestor in the search tree: More specifically, **PT** register is updated as follows:

- First, **PT** register is modified upon encountering a generator tabled subgoal. This is performed by the tabletry and tabletrysingle instructions, after the creation of the generator choice point frame, and the value of **PT** register is set to the address of that choice point.

- Secondly, as is done for all other registers of the WAM, **PT** register is restored to its earlier value in the event of backtracking (through the retry, trust, tableretry, and tabletrust instructions, as well as the instructions of Section 4.5.3 that backtrack through answer tries).

- Finally, **PT** register should be also be restored in forward continuations of tabled predicates. To illustrate, let

$$p(\overline{a}) \ :- \ l_1(\overline{b_1}), \ldots, l_i(\overline{b_i}), \ldots, l_n(\overline{b_n}).$$

  be the clause under execution, and assume that the subgoal $t_i(\overline{b_i})$ of literal $l_i(\overline{b_i})$ is a tabled subgoal. Observe that the call to $t_i(\overline{b_i})$ will alter the value of **PT** register if $t_i(\overline{b_i})$ is a generator node (program clause resolution will then be used). To compensate for the possible change, upon the success of $l_i(\overline{b_i})$, (for example through the return of an answer for $t_i(\overline{b_i})$), the value of **PT** register is restored to the value saved in the *PTreg* field of the choice point frame for $l_i(\overline{b_i})$. This restoration is unnecessary if $t_i(\overline{b_i})$ is a non-tabled subgoal. Therefore the restoration of **PT** register during forward execution should only be performed by the new_answer, answer_return, and negation_resume (which will be introduced in Section 5.4) instructions.

5.4 Suspending and Resuming Negative Literals

In order to implement the NEGATION SUSPENSION and NEGATION SUCCESS operations, mechanisms must be introduced to suspend clauses on negative literals and to resume the computation of these clauses when the subgoal of the suspended negative literal is completed. Note that facilities in the SLG-WAM for definite programs that handle active subgoals and support the return of answers to suspended subgoals already support efficient suspend and resume operations.

The NEGATION SUSPENSION operation is implemented through a C-level builtin, called negation_suspend/1. This builtin suspends the computation by laying down a *negation suspension frame*. This frame, which resides on the choice point stack, saves the execution environment for the suspended computation in a manner similar to saving suspended environments in active subgoal frames. In addition to the usual information of WAM environments, this frame also contains a pointer to the *Subgoal* that is being negatively suspended, and, for non LRD-stratified programs, a *StratStatus* field whose use is explained in the following chapter. Like active subgoal frames, negation suspension frames are chained together

EBreg	Environment Backtrack Point
Hreg	Top of Global Stack (Heap)
TRreg	Top of (Forward) Trail Stack
CPreg	Return point of suspended literal
Ereg	Parent Environment
PTreg	Parent Tabled Choice Point
Subgoal	Frame of Suspended Subgoal
*StratStatus**	Stratification Status of the Suspension
PrevNSF	Previous Negation Suspension Frame

Figure 28: *Format of Negation Suspension Frames.*

(using the *PrevNSF* field) and can be accessed through the *NSFs* field of the subgoal frame (see Figure 10). The format of *negation suspension frames* is shown in Figure 28.

Execution will resume to the environment saved by negation_suspend/1 builtin when the subgoal gets completed (Section 5.6 provides full details on when this is done). When completing failed subgoals (those for which the negative literal will succeed) the completion instruction creates *negation resume choice points* chained together on the choice point stack. Upon backtracking to these choice points, the engine will use the trail to switch execution to the environments of the suspended computations (using the completion suspension frame) and continue with forward execution of the suspended clause. The actual implementation of the negation_resume instruction (sketched in Figure 29) resembles that of answer_return except that no actual answer need be returned, so only the environment of the negatively suspended computation is restored.

```
/* current breg points to a negation resume choice point (NRCP) */

Instruction negation_resume
    NSF := NRCP_NSF(breg);
    restore_bindings(old_breg,NSF);     /* restore suspended environment */
    Restore all values of WAM registers as these were saved in the NSF;
    if (there is another negation suspension frame NSF' in the NSF list)
        NRCP_NSF(breg) := NSF';      /* NSF' will be picked up next */
    else   /* backtrack out of negation_resume */
        breg := NRCP_PrevCP(breg);
    set the program pointer P to the continuation pointer CP;
```

Figure 29: *The* negation_resume *instruction.*

5.5 Implementation of a Predicate for LRD-Stratified Negation

The predicate `tnot/1` implements well-founded negation in the XSB system. Figure 30 shows its implementation restricted to the class of LRD-stratified programs. A consequence of the SLG operations defined in Section 5.1 is that any negation suspensions on subgoals that obtain answers could be removed by the early completion operation. This removal causes the failure of the suspended clauses, effectively implementing a NEGATION FAILURE operation. Thus, upon the COMPLETION of subgoals that still have negation suspensions on them, the selected literal of the clauses suspended on these subgoals always succeeds and can simply be removed. Therefore, the following invariant is maintained:

Invariant II *In the SLG-WAM for LRD-stratified programs, the completion of subgoals initiates only* NEGATION SUCCESS *operations.*

Figure 30 sketches the implementation of `tnot/1` using low-level builtins. Note that ground subgoals with answers are determined as completed by the early completion operation. `tnot/1` therefore calls the `negation_suspend/1` builtin only if the subgoal is incomplete

```
tnot(S) :-
    ( ground(S) →
        ( subgoal_not_in_system(S), call(S), fail
        ; (is_complete(S) → has_no_answers(S)
          ; negation_suspend(S), true   /* if execution reaches here, S */
          )                             /* is completed with no answers */
        )
    ; error("Flounder: subgoal ", S, " is not ground")
    ).
```

Figure 30: *The implementation of tabled negation for LRD-stratified programs.*

and has no answer. Later, the computation resumes (to `true`) only if the completed subgoal has no answers. As a final comment, we note that the groundness test of `tnot/1` can be efficiently implemented in the SLG-WAM framework by an operation that checks whether the number of variables in the substitution factor of the arguments of the subgoal, that is stored immediately below the generator choice point for S, is zero (see Figure 15 and the discussion on substitution factoring in Section 4.5.2).

5.6 Completion in LRD-Stratified Programs

Selection of a negative literal may in certain cases cause an *active* clause to *suspend* on another negative subgoal in the same ASCC, complicating the completion algorithm. If the engine is to maintain a fixed computation rule for LRD-stratified programs, it must correctly order the completion of subgoals and the resumption of the suspended computations. The following example illustrates some of the difficulties that are encountered when ASCCs contain negative dependencies and is used throughout this section to explain the COMPLETION operation.

Example 5.6.1 Let P be the program in Figure 31 for which the query `?- p(a)` is to be evaluated. Note that since there is only one predicate, `p/1`, P is not modularly stratified for any selection order. It is, however, LRD-stratified. The evaluation of `p(a)` causes cascading

```
:- table p/1.

p(a) :- p(b), ¬p(d).

p(b) :- p(c).
p(b).
p(b) :- ¬p(a).

p(c) :- p(b), p(g).

p(d) :- p(e), p(f).

p(e).

p(f) :- ¬p(c), p(d).

p(g) :- p(c).
p(g) :- ¬p(e), ¬p(g).
```

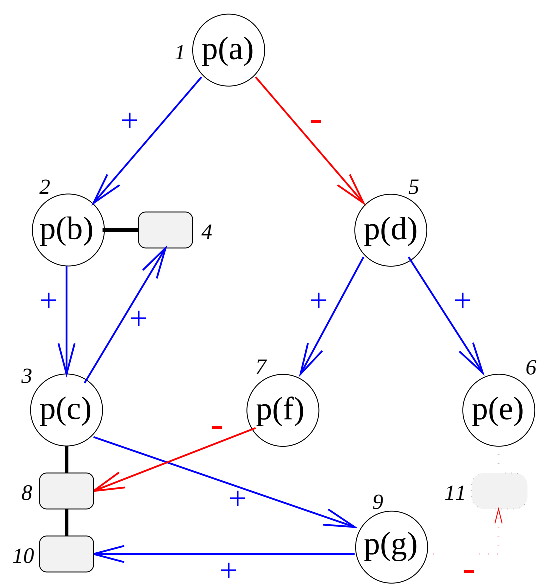


Figure 31: *A LRD-stratified program and the SDG for the query* `?- p(a)`.

suspensions, as seen from an elaboration of its *SDG* in which circled nodes denote *generator* nodes, while boxes denote corresponding *active* or *suspended* nodes. The *SDG* represents the state when the literal under execution is `¬p(e)` (shown underlined in the last clause of P) when all clauses and literals but those in boxes have been selected. The numbers next to the nodes of the *SDG* depict the order in which these subgoal instances are encountered by the SLG-WAM. The answer derived for `p(b)` has already been returned to both the `p(a)` and `p(c)` clauses by means of the ANSWER RETURN operation. The clauses for `p(a)` and `p(f)` are suspended on the completion of `p(d)` and `p(c)` respectively. Note that the selected literal `¬p(e)`, and consequently the second clause for `p(g)`, need not be suspended on the completion of `p(e)` because `p(e)`, which currently has an answer, is determined to be completely evaluated by early completion. Thus, a NEGATION FAILURE operation is immediately applicable to the clause. This operation will remove the dependencies due to the second clause of `p(g)` from the *SDG* along with the dotted part of Figure 31(b). Since there are no more clauses for subgoals `p(b)` through `p(g)` that are not suspended, the engine fails back to the choice point for `p(b)`, and schedules a **completion** instruction. The effect of executing this **completion** instruction will be outlined in Example 5.6.2. ■

In general, to find a correct order of completing subgoals in an SLG system, positive as well as negative dependencies between subgoals must be taken into account. Completing all subgoals that have negation suspensions and are not immediately suspended on other subgoals is not a correct completion strategy. For example, notice that in Example 5.6.1 the only subgoals that have negation suspensions on them are `p(c)` and `p(d)`. Choosing to complete these subgoals first, and release their suspensions, is not a strategy that will yield correct results.

For simplicity of presentation we will present the algorithm that implements the COMPLETION operation as working exclusively on the completion stack. In reality, it needs information both from the choice point stack, and from the subgoal frames.

For LRD-stratified programs, each completion stack frame is augmented with two extra fields, SDG_Edges and SDG^T_Edges, that are used for the edges of the SDG and SDG^T stemming from each subgoal in the stack. In actuality, the edges of the SDG and SDG^T are constructed lazily from the choice points whenever needed. The XSB implementation of the SLG-WAM constructs these edges whenever the subgoals of the current $ASCC$ contain negation suspensions, thus avoiding this construction for definite programs, an important optimization step as will be seen in Chapter 7. Figure 32 shows completion stack states for the program of Example 5.6.1 before and after the removal of completion stack frames

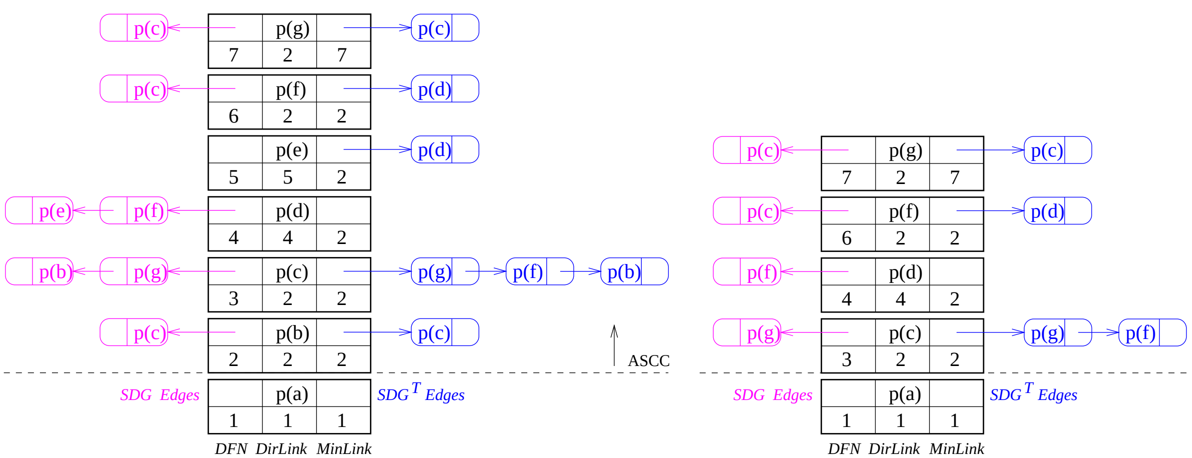


Figure 32: *Completion Stacks when evaluating the program of Example 5.6.1.*

of subgoals through early completion. The edges of SDG and SDG^T (shown as adjacency lists) are restricted to the subgoals of the $ASCC$ that has subgoal `p(b)` as its leader. This $ASCC$ is also indicated in the same figure.

Recall that rather than eliminating the generator choice point frame, either the **tabletrust** or the **new_answer** instruction (in the case of early completion) places a **completion** instruction in the *FailCont* cell of its own choice point. Upon later backtracking to this choice point, the

```
Instruction completion
0      subgoal := TCP_subgoal(breg);
1      schedule_answers(subgoal);                    /* pre-completion part */
2      if (subgoal is a leader of an ASCC, A)  /* according to Definition 5.6.1 */
2.1      if (there are no negation suspensions on subgoals in A)
2.1.1      Mark as completed all subgoals in A;
2.1.2      Reclaim the stack space of subgoals in A and adjust the freeze registers;
         else
           Let E be the set of subgoals of the ASCC that were early completed;
2.2        I := independent_scc(subgoal);      /* see Figure 34 */
2.3        For each subgoal s ∈ I
2.3.1        Mark subgoal s as completed;
2.3.2        if (there are negation suspensions on s)
               Schedule negation_resume instructions for each of these suspensions;
2.3.3          if (there exists a subgoal s' ∈ I that is suspended on s)
2.3.4            Warn that the program is not LRD-stratified;
2.4        C := E ∪ I;
2.4.1      Compact the completion stack by removing the frames of subgoals in C;
2.4.2      Reclaim the stack space of subgoals in C and adjust the freeze registers;
3      else breg := TCP_PrevCP(breg);          /* PrevCP field of tabled CP */
4      fail;
```

Figure 33: *The* completion *instruction (for LRD-stratified Programs).*

completion instruction will determine whether the SCC in which this subgoal participates is completely evaluated. As can be seen from the instruction's pseudocode in Figure 33, its first action is to get the table entry that is associated with the current generator tabled choice point. After all unresolved answers of this subgoal have been returned to all appropriate consuming nodes, a check is made to determine whether this subgoal is a leader of an $ASCC$. This check is performed using Definition 5.6.1 which reflects the fact that completion may selectively complete subgoals of an ASCC. Similarly to the condition of Definition 4.6.1, the following condition ensures that the subgoal will not be considered a leader if it has direct or indirect dependencies lower into the stack.

Definition 5.6.1 (*Leader*) A subgoal S is called a *leader* of an $ASCC$ iff the completion stack frame associated with S satisfies the condition:

$$DFN(S') < min(DirLink(S), MinLink(S))$$

where S' is the subgoal that is immediately below S in the completion stack. □

If the subgoal is not a leader, the action of the completion instruction is to simply backtrack to the previous choice point. If, however, the subgoal is a leader of an $ASCC$ that does not

---

```
/* Let SDG be the subgoal dependency graph of the program-query pair, */
/* let subgoal be a leader of an ASCC of SDG, and let V' be the set of */
/* subgoals in this ASCC when early completed subgoals are excluded. */
/* Let SDG' be the subgraph of SDG induced by V'. */
/* Finally, let SDG'^T be the transpose of SDG'. */

Procedure independent_scc(subgoal)
    Starting from any subgoal in V', perform a depth-first search of SDG'^T
        to find the subgoal s ∈ V' with the maximum finish time;
    Starting from s, perform a depth-first search of SDG' to obtain
        the set of subgoals I ⊆ V' that are in the same depth-first tree as s;
    return I;        /* I is an independent SCC */
```

Figure 34: *Pseudocode for a procedure that finds an Independent SCC.*

contain negative links, all subgoals in the $ASCC$ can be completed and their space can be reclaimed. If the subgoal is a leader and the $ASCC$ does contain a negative link the engine must find an independent SCC using the method that is sketched below.

Because the SDG is effectively represented by information in the completion stack, finding an independent SCC is straightforward (see Figure 34). Using the completion stack, a subgraph SDG' of SDG is derived for those subgoals having the currently completing *subgoal* as their leader. The algorithm then performs a combination of a topological sort and an SCC computation of a directed graph [18]. It essentially computes the set of subgoals that constitutes a vertex of the *component graph* of SDG that has no incoming edges. The procedure does not need to examine subgoals that were completely evaluated through early completion, since the stack-based completion algorithm would have marked these as completed.

Definition 5.6.2 (*Component Graph*) A *Component Graph* of a graph $G = (V, E)$ is a graph $G^{SCC} = (V^{SCC}, E^{SCC})$ where V^{SCC} contains one vertex for each strongly connected component of G, and E^{SCC} contains the edge (u, v) if there is a directed edge from a vertex in the strongly connected component of G corresponding to u to a vertex in the strongly connected component of G corresponding to v. □

Intuitively, G^{SCC} is obtained by shrinking each strongly connected component of G to a single vertex. The edges of the component graph express dependencies between the SCCs of G. It can be proven that the G^{SCC} is a directed acyclic graph. Moreover, the notion of an independent component can be characterized in terms of the component graph: a strongly connected component C is independent if and only if C has no outgoing edges in G^{SCC}. Figure 35 shows the component graphs of the SDG Figure 31(b) where early completed

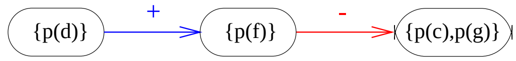


Figure 35: *Component graphs of the subgoals in the ASCC of Figure 32.*

subgoals have been excluded. The correctness of procedure independent_scc follows from the proposition below.

Proposition 5.6.1 *The SCC containing the subgoal with the greatest finish time obtained by a depth-first search of SDG^T is an independent SCC.*

Furthermore, the correctness of the completion algorithm follows from the proposition below, which can be proven using the formalism of SLG_O automata in [68].

Proposition 5.6.2 *Let S be the completion stack in an evaluation of a LRD-stratified program, and let L be the leader in the completion stack with greatest DFN, and assume that the subgoals in $ASCC(L)$ have performed all program and answer clause resolution to non-suspended clauses. Finally, let C be the set of subgoals completed by the algorithm in Figure 33. Then,*

1. *C will be non-empty;*
2. *No subgoal will be in C unless it is completely evaluated.*

The subgoals of the independent SCC are marked as completed and negation_resume instructions are scheduled for all negative literals that were suspended on the completion of these subgoals. This scheduling is done by modifying the **B** register to point to a *negation resume choice point* (Section 5.4) which is responsible for initiating these instructions (see Figure 29). In addition, a run-time check on whether the program is LRD-stratified can be performed by the completion instruction. The program-query pair under execution is not LRD-stratified if and only if there exists a negative dependency among the subgoals of an independent SCC. This condition is checked in step 2.3.3. Finally, the completion stack is compacted, the remaining choice points are (re)chained through their *PrevCP* field, and, if possible, the stack space of the subgoals that are marked as completely evaluated is reclaimed.

Example 5.6.2 Continuing Example 5.6.1, the COMPLETION operation for the $ASCC$ led by `p(b)` finds subgoals `p(c)` and `p(g)` to be an independent SCC, and marks them as completely evaluated. The completion frames of these subgoals, as well as those of subgoals `p(b)` and `p(e)` that were early completed, are removed from the completion stack. Also, their completion initiates a NEGATION SUCCESS operation that unsuspends literal ¬`p(c)` in the clause of `p(f)`. When this literal is resumed, computation continues with literal `p(d)` in the body of `p(f)` which modifies the subgoal dependency graph, and makes the state of the completion stack to be that shown in Figure 36. Computation fails back to the

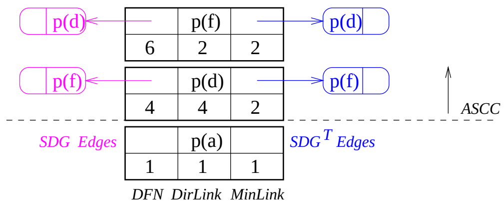


Figure 36: *Completion Stack after subgoals* `p(b)`, `p(c)`, `p(e)`*, and* `p(g)` *are completed.*

choice point for `p(d)`, and a **completion** instruction is executed next. Using the condition of Definition 5.6.1, `p(d)` is found to be a leader of another $ASCC$ (note that subgoal `p(f)` is not a leader). Subgoals `p(d)` and `p(f)`, which are now mutually dependent and consequently part of the same independent SCC, are both marked as completely evaluated. Finally, literal $\neg$`p(d)` in the body of `p(a)` gets unsuspended (using a **negation_resume** instruction) and succeeds, which in turn activates the early completion of subgoal `p(a)`. ■

Even though not shown in the above example, in general the approximation of the strongly connected components kept by the completion stack may considerably change as a result of resuming suspended computations, since **negation_resume** can initiate new, untried computation paths.

The stratification check of step 2.3.3 of the completion algorithm for LRD-stratified programs is the starting point for the evaluation of programs that are not fixed-order dynamically stratified, and consequently require use of the SLG DELAYING and SIMPLIFICATION transformations. How the latter operations are performed in the framework of the SLG-WAM is the subject of the following chapter.

Chapter 6

The Abstract Machine for Non LRD-Stratified Programs

In order to evaluate according to the Well-Founded Semantics programs which are not LRD-stratified, four main additions are necessary to the engine of the previous chapters. A mechanism for detection and handling of loops through negation using the DELAYING transformation, a mechanism for representation, maintenance, and manipulation of conditional answers, the ability to detect when the SIMPLIFICATION and ANSWER COMPLETION operations should be triggered, and finally the procedures that perform all these operations. Naturally, these mechanisms should be supported by appropriate data structures of the abstract machine, and their addition should not significantly affect the performance characteristics of programs which are either definite or fixed-order dynamically stratified. We present a detailed description of these additions in turn; we first, however, present through an example the additional tabling operations that are involved in the evaluation of these programs.

6.1 Tabling for Non LRD-Stratified Programs

In the evaluation of normal programs, a set of tabled subgoals may depend upon one another through negation in such a way that none can be determined to be completely evaluated before the rest. To allow tabled evaluation to proceed, SLG applies a DELAYING transformation to ground negative literals involved in negative loops, so that the remaining literals in the body of clauses can be resolved. The delayed literals are maintained in a *delay list* so that each node, including answers, is represented as:

$$answer\_template \;:\!-\; [delay\_list]\; goal\_list.$$

If the *goal_list* is empty, then the node is an answer which is added in the table through the NEW ANSWER operation. In SLG, each node is associated with the subgoal S that forms the root of the tree, so that *answer_template* is subsumed by S, and if the node is an answer, we will speak of the *answer substitution* which is the most general unifier of the *answer_template* and S. Answers in the system whose delay list is empty are termed *unconditional*, while answers with a non-empty delay list are termed *conditional*. We refer to the elements of the *delay_list* as *delayed literals*. An answer is called *supported* if none of the literals in its delay list is false; otherwise it is *unsupported*. Unconditional answers are always supported. Note that there is a one-to-one correspondence between an *answer_template* and an answer substitution; consequently, we may group properties of answers according to answer substitution. We also speak of an *unconditional* (supported) *answer substitution* η_S in an SLG system $\mathcal{S}$ if there exists in $\mathcal{S}$ an unconditional (supported) answer with answer substitution η_S. Otherwise, the answer substitution is *conditional*. Finally, recall that a subgoal S *succeeds* if it has an unconditional answer that is a variant of S, and it *fails* if it is completely evaluated with no answers.

Delayed literals within conditional answers may be removed from the body of clauses through SIMPLIFICATION operations. We further discuss DELAYING and SIMPLIFICATION, along with other aspects of SLG, through the following example.

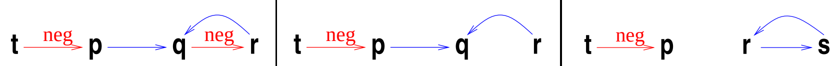
Example 6.1.1 We present in detail the evaluation of a query `?- t.` with respect to the program of Figure 37 where all predicates are tabled (i.e., evaluated using SLG). We depict

```
:- table_all.

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

Figure 37: *A Normal Logic Program (non LRD-stratified).*

an SLG clause with status *status* of an SLG tree of subgoal S as S : *status* : A `:-` *Body*, where the head of the clause, A, is an *answer_template*. In the first stage of the evaluation, the system in Figure 38(a) is produced. In this system, the leftmost literal of the first four clauses has been selected. Clauses waiting for answers to be returned to them have the status *act* (*active*), those waiting for completion of a negative literal have the status *susp* (*suspended*). As can be seen from its accompanying *SDG*, the evaluation has encountered a negative loop containing `q` and `r`. Together, both subgoals form an SCC which is also independent. In order to determine the truth value of `q` and `r` in the well-founded model, the computation rule must be broken so that other literals in clauses for `q` and `r` may be resolved. The negative literals involved in the independent SCC are delayed, producing the system in Figure 38(b) in which delayed literals are shown in a delay list. At this stage the clause `q`:*act*:`q :- [¬r].` has no more literals to resolve and becomes a conditional answer, `q`:*ans*:`q :- [¬r].` (through the NEW ANSWER operation). This conditional answer can be returned to the clause for `r` or to that for `p`. We assume that it is first returned to `p`

(a)	(b)	(c)	(d)
t:*susp*:t :- ¬p. p:*act*:p :- q. q:*susp*:q :- ¬r. r:*act*:r :- q, s.	t:*susp*:t :- ¬ p. p:*act*:p :- q. q:*act*:q :- [¬r]. r:*act*:r :- q, s.	t:*susp*:t :- ¬p. p:*ans*:p :- [q]. q:*ans*:q :- [¬r]. r:*act*:r :- [q],s. s:*act*:s :- r.	failed(t). p:*ans*:p. q:*ans*:q. failed(r). failed(s).



(a) (b) (c) (d)

Figure 38: *Selected SLG systems, and their SDGs for the query* ?- t.

(through the ANSWER RETURN operation) producing another conditional answer, and to r which then calls s and afterwards, recursively, itself. The resulting system is shown in Figure 38(c). At this stage, s and r, which are in a positive loop, have been completely evaluated and can be completed, with both s and r failed. Upon the completion of r, the answer q:*ans*:q :- [¬r]. should be made unconditional, since ¬r is now true. SLG addresses situations such as this through the SIMPLIFICATION operation which simplifies away delayed literals of conditional answers. Using a SIMPLIFICATION operation, ¬r is removed from the delay list for q making the answer q unconditional. This unconditional answer enables a further SIMPLIFICATION operation in the answer for p. Finally, since p has succeeded the suspended clause for t fails (through a NEGATION FAILURE operation), leading to the final system of Figure 38(d). ■

We summarize the actions of Example 6.1.1 in handling negative loops: DELAYING can be thought of as a mechanism for dynamically changing the left-to-right computation rule. As seen from the SDG of Figure 38(b), delaying a literal may also avert dependencies through negation. As shown by the failure of r, DELAYING may also allow a clause that creates a cyclic negative dependency to fail based on the falsity of a literal in its body. Such failures trigger SIMPLIFICATION operations. In general, delayed literals that are true should be deleted from answers, and answers with delayed literals that are false should be deleted from the system.

When resolution occurs for an *active* or *suspended* node, any variable bindings accumulated in the answer head are propagated through unification. However, delayed literals themselves are not propagated. Instead, positive delayed literals are created as placeholders for propagating truth values of delayed ground negative literals. In Figure 38(c), the active clause for p uses the placeholder q, rather than propagating the delayed list [¬r].

This use of positive delayed literals guarantees a polynomial representation of answers of queries that may otherwise have an exponential number of answers with only ground negative literals in answer bodies (see [14] for an example of that situation). In the final SLG system, each delayed literal in the body of an answer corresponds to the head of some *conditional* answer of that subgoal; i.e. an answer with delayed literals in its body. The set of answers resulting from the evaluation of a query Q with respect to a program P can be viewed as a *residual program* of the program-query pair $\langle P, Q\rangle$. This residual program consists of all answers of the query plus the rules of all head atoms upon which the answers of the query depend directly or indirectly through delayed literals in answer bodies. Furthermore, the residual program, whose subgoal dependencies have been simplified according to the well-founded semantics, can be further processed to yield results in other forms of non-monotonic reasoning such as stable models [29], or the well-founded semantics with explicit negation [43].

6.1.1 Non LRD-Stratified SLG-WAM: Overview

Example 6.1.1 brings up a number of yet unaddressed issues that are involved in implementing the well-founded semantics in the SLG-WAM framework.

Detecting and Resolving Negative Loops As part of implementing the COMPLETION operation for LRD-stratified programs, Section 5.6 has already sketched a possible method of detecting negative dependencies between incomplete subgoals. Section 6.2 discusses how this negative loop check is extended to normal programs, and Section 6.3 discusses the implementation of the `tnot/1` predicate that resolves negative literals in normal programs.

Storing Conditional Answers in Tables In general, DELAYING associates conditions with answer substitutions. Section 6.4.3 addresses how answer tables in the form of tries can be extended to store delay lists and to support SIMPLIFICATION.

Manipulating Delay Lists Delay lists occur not only in answers, but in the heap for *active* and *suspended* clauses. Section 6.4 discusses the representation of delay lists, and their maintenance under backtracking, suspend and resume.

Simplification Operations The table space must also be extended to efficiently perform simplification when the truth value of a literal in the delay list of an answer becomes known, and so that these SIMPLIFICATION operations can be propagated if necessary. This is the subject of Section 6.5

6.2 Detection and Handling of Negative Loops

As shown by Example 6.1.1, the evaluation of arbitrary normal programs may encounter loops through negation. For the correct evaluation of such programs, a negative loop detection mechanism is required. Fortunately, in the SLG-WAM the same dependency mechanism that detects completion of tabled subgoals can also be used to detect SCCs containing loops through negation. The actual operation of DELAYING negative literals involved in negative loops is performed by the negation_resume instruction, and is described in the following sections.

6.2.1 Completion in Normal Logic Programs

The completion SLG-WAM instruction for the evaluation of normal logic programs under the well-founded semantics is presented in Figure 39. To the extent that was possible, its structure follows that of the completion instruction used for the evaluation of fixed-order dynamically stratified programs presented in Figure 33, and the reader is encouraged to contrast them.

As described in Section 5.3, the Subgoal Dependency Graph of the program-query pair is embedded in the choice point stack of the SLG-WAM, and the completion stack already contains its stack-based approximation. If there are no negation suspensions, completion can safely complete subgoals using this stack-based approximation. Otherwise, subgoals should be completed based on exact dependencies, and possible negative loops have to be broken through DELAYING. Four steps of the completion instruction (indicated in Figure 39 by numbers) perform this.

The first step lazily constructs the dependency graph of the system in order to find an *independent strongly connected component* I, and to determine if there is a negative loop among subgoals in I. This is done according to procedure independent_scc of page 77, with the only exception being that dependencies due to delayed negative literals are excluded from the subgoal dependency graph. The second step performs a left-to-right dynamic stratification check, while at the same time it classifies the suspensions of subgoals in I. Recall that each negation suspension frame of a subgoal s contains information about the nearest tabled subgoal s' that is suspended on the completion of s. If both s and s' are in I, there is a direct negative dependency between subgoals in I. Since the subgoals of I form a strongly connected component, the subgoal dependency graph induced by I contains an unbroken loop through negation. All negation suspension frames that are directly involved in the negative loop have their *StratStatus* field (cf. Figure 28) set to *delayed*, they are removed from the subgoal dependency graph and are collected in a set NS. Note that NS consists of those suspensions for which both subgoals are in I. If NS is empty, then the subset of the program examined so far is LRD-stratified, all subgoals of I are completed, and all negation suspensions of these subgoals are resumed in the way described in Section 5.4. If, however, NS is not empty, the direct negative dependencies of the independent SCC are broken by applying DELAYING operations on them. This is done by scheduling negation_resume instructions for each suspension in NS. Recall that these negation suspension frames were marked as *delayed* in step 2; this information is used by the negation_resume instruction. Note that when there are negative loops, no subgoal is completed. The completion of subgoals in I is postponed for a future invocation of the completion instruction; when all possible DELAYING operations have been performed, and there are no more unbroken loops through negation. Finally, the last step reclaims the stack space of subgoals that were completed, and compacts the completion stack to preserve Invariant I of the SLG-WAM (cf. page 62).

```
Instruction completion
  subgoal := TCP_subgoal(breg);
  schedule_answers(subgoal);              /* pre-completion part */
  if (subgoal is a leader of an ASCC, A)  /* according to Definition 5.6.1 */
    if (there are no negation suspensions on subgoals in A)
      For each subgoal s ∈ A
        Mark subgoal s as complete;
        if (s fails and s has negative IDEs)
          Initiate a simplify_neg_fails(s) SIMPLIFICATION;
      Reclaim the stack space of subgoals in A and adjust the freeze registers;
    else
      Let E be the set of subgoals of the ASCC that were early completed;
1.    I := independent_scc(subgoal);      /* see Figure 34 */
      NS := ∅;                 /* set of non-stratified negation suspensions */
2.    For each subgoal s ∈ I   /* Classify the suspensions of subgoals in I */
        For each negation suspension ns of s
          if (the subgoal s' that is suspended on s is in I)
            Mark ns as involved in a loop through negation (delayed);
            Remove ns from the list of negation suspensions of s;
            NS := NS ∪ {ns};
3.    if (NS = ∅)              /* Perform completion as in LRD programs */
        For each subgoal s ∈ I
          Mark subgoal s as complete;
          if (s fails and s has negative IDEs)
            Initiate a simplify_neg_fails(s) SIMPLIFICATION;
          if (there are negation suspensions on s)
            Schedule negation_resume instructions for each of these suspensions;
        C := E ∪ I;
      else                     /* There are unbroken negative loops */
        For each negation suspension ns ∈ NS   /* initiate DELAYING */
          Schedule a negation_resume instruction for ns;
        C := E;
4.    Compact the completion stack by removing the frames of subgoals in C;
      Reclaim the stack space of subgoals in C and adjust the freeze registers;
  else breg := TCP_PrevCP(breg);
  fail;
```

Figure 39: *The* completion *instruction (for Normal Logic Programs).*

6.3 Implementation of a Predicate for Well-Founded Negation

As mentioned in Section 5.5, the predicate `tnot/1` implements tabled negation, according to the well-founded semantics, in the XSB system. Figure 40 sketches its implementation using C-level builtins. When the negation involves a ground subgoal that is completed, the builtin negate_truth_value/1 fails, succeeds, or delays depending on whether the subgoal succeeds, fails, or has only conditional answers, respectively.

```
tnot(S) :-
    ( ground(S) →
        ( subgoal_not_in_system(S), call(S), fail
        ; ( is_complete(S) → negate_truth_value(S)
          ; negation_suspend(S), tnot_resume(S)
          )
        )
    ; error("Flounder: subgoal ", S, " is not ground")
    ).
```

Figure 40: *The implementation of tabled negation for normal logic programs.*

A more complex case of tabled negation occurs when the subgoal is still incomplete. Then, the negation_suspend/1 builtin suspends the current computation (as explained in Section 5.4), which will later be reinvoked when more is known about the truth value of the subgoal.

Three outcomes can occur for a literal that is negatively suspended: it can succeed, fail, or its subgoal can get completed with conditional answers only. Negation suspensions of subgoals that succeed are removed by the *early completion* operation (cf. Section 5.5). As a result, the tnot_resume/1 builtin will never be called, so that the associated suspended clauses effectively fail. Otherwise, the completion instruction will schedule negation_resume instructions, by creating *negation resume choice points* chained together on the choice point stack. Upon backtracking to these choice points, the engine resumes the environment of the suspended computation (using the negation suspension frame), and continues execution of the suspended clause. Operationally, this occurs when:

1. unsuspending on determining that the subgoal is completely evaluated;
2. DELAYING the negative literal before the completion of its subgoal in order to break a loop through negation.

In the first case, the subgoal will either be failed, or will have conditional answers only. If the subgoal is failed, tnot_resume/1 will do nothing, effectively resolving away a negative literal for a failed subgoal. If the subgoal has conditional answers, **tnot_resume/1** will add the conditional negative literal to the delay list. In the second case, the **negation_resume** instruction was scheduled to break a loop through negation, and the *StratStatus* field of the negation suspension frame is marked as *delayed* (by the COMPLETION instruction of the previous section). Once again, **tnot_resume/1** will add the negative literal to the delay list, as if it were completed with conditional answers.

6.4 Delaying and Handling of Conditional Answers

6.4.1 Delay Lists and Delay Elements

As discussed in Section 6.1 delay lists consist of literals which are either undefined in the well-founded moded, or literals in the body of a clause whose truth value is yet undetermined in the current SLG system. When and if the truth value of these literals becomes known, SIMPLIFICATION operations will either remove the literals from the delay list, or dispose of the clauses. The representation of delay lists and delay elements in the SLG-WAM must therefore support several operations: adding elements to the delay list in the course of resolution; copying the delay list into the table, and the various forms of SIMPLIFICATION.

At an operational level, a delay list can be thought as an extra argument in tabled predicates. At subgoal call the argument is initialized to an empty delay list represented by a variable; during program and answer clause resolution this variable may get bound to a list of literals. In terms of the table, the binding of the delay argument is treated as part of an answer substitution and copied into the answer table if the derivation is successful. Obtaining a representation of delay elements to support the above operations is complicated by changes made to the variable bindings of a literal in the course of resolution. Consider the operational detail of delaying a literal for a clause C. First, a selected literal is called during the course of computation: for purposes of illustration let this literal be $p(X,Y)$. Then, suppose $p(X,Y)$ has a conditional answer: $p(a,Y)$:− $[delay\_list]$. Answer clause resolution will propagate the binding $X = a$ into C, and $p(a,Y)$ is added to C's delay list. Finally, suppose further resolution of C adds the binding $Y = b$. How should the delayed literal be represented within the delay list? As $p(X,Y)$, as $p(a,Y)$ or as $p(a,b)$? In the SLG-WAM, efficient support of the above operations requires the delay element be represented by a triple of the form:

$$\langle Subgoal\_Id, Literal, Answer\_Substr\_Id\rangle$$

where:

$Subgoal\_Id$ indicates the subgoal of the delayed literal *e.g.* $p(X,Y)$;

Literal indicates the delayed literal with all bindings accumulated during the course of resolution *e.g.* $p(a,b)$; and

Answer_Subst_Id either indicates the answer substitution if the delayed literal is positive, *e.g.* $p(a, Y)$, or contains the marker $\neg$ if the delayed literal is negative.

In terms of SLG-WAM data structures, *Subgoal_Id* is a pointer to a *subgoal frame* (see Section 6.5.4), while *Answer_Subst_Id* is a pointer to a node of the answer trie that uniquely identifies the answer substitution (Section 6.4.3 provides details of answer tries in the presense of conditional answers). A null pointer is used to denote the $\neg$ indicator. Note that the only part of the *Literal* that has to be explicitly stored is the substitution for the variables in the answer (for a positive delay element), or in the subgoal (for a negative delay element). Also note that even though SIMPLIFICATION may delete elements from the delay lists in tabled answers it will never change the bindings for these delay elements, including information in their *Literal* field.

Example 6.4.1 Consider the evaluation of query $Q = \texttt{p(X)}$ with respect to the non LRD-stratified program of Figure 41(a), where tabled evaluation is used for all predicates. This query produces the subgoals shown in Figure 41(b) together with their answers. During the evaluation of Q, DELAYING is needed for subgoal s, and the ability to return conditional answers to active subgoals is needed for the evaluation of subgoals $q(X, Y)$ and $p(X)$. No SIMPLIFICATION is possible. Answers consist of two parts: an atom that is an instance of

```
:- table_all.

p(X) :- q(X,Y), r(Y).
p(a) :- s.
q(a,Y) :- s.
s :- ¬ s.
r(b).
r(c).
```

(a)

Subgoal	*Answer*		*Answer_Id*
$p(X)$	p(a)	$[\langle q(X,Y), \texttt{q(a,b)}, q(a,Y)_1\rangle]$	$p(a)_1$
	p(a)	$[\langle q(X,Y), \texttt{q(a,c)}, q(a,Y)_1\rangle]$	$p(a)_2$
	p(a)	$[\langle s, \texttt{s}, s_1\rangle]$	$p(a)_3$
$q(X,Y)$	q(a,Y)	$[\langle s, \texttt{s}, s_1\rangle]$	$q(a,Y)_1$
s	s	$[\langle s, \texttt{s}, \neg\rangle]$	s_1
$r(Y)$	r(b)	$[\,]$	$r(b)_1$
	r(c)	$[\,]$	$r(c)_1$

(b)

Figure 41: *A program requiring* DELAYING *and tables created for* `?- p(X).`

the subgoal containing substitutions for its variables, and a delay list that indicates the conditionality of the answer. Observe that the same answer substitution can be conditional on many different sets delayed literals. Section 6.4.3 illustrates how the SLG-WAM supports this many-to-many relationship in an efficient manner. Also, note the delay list of the answer identified as $p(a)_1$.[1] Its single element records information about the subgoal in the body of $p(X)$ that was delayed ($q(X, Y)$), the answer of that subgoal which was conditionally returned ($q(a, Y)_1$), and the form of the delayed literal (`q(a,b)`) at the time when the answer $p(a)_1$ was inserted in the table. Finally note that answers of possibly different subgoals can be conditional on the same sets of delayed literals (as are answers $p(a)_3$, and $q(a, Y)_1$ in this example). ■

[1] Throughout this chapter, we identify subgoals by atoms, and answer substitutions by subscripted atoms. We use the terms answer and answer substitution interchangeably whenever there is a unique answer for a given answer substitution.

For notational convenience, we will present answers by indicating their *answer substitution*; i.e. a set of substitutions for the variables in the subgoal. Also, we will usually use the shorthand $\langle \neg L \rangle$ to denote a negative delay element of the form $\langle G, L, \neg \rangle$, since in a non-floundering evaluation, G must be ground and so $G = L$. Variations of the program in the following example will be used throughout this chapter to describe the data structures used to implement DELAYING and handling of conditional answers.

Example 6.4.2 Consider the evaluation of query `?- p(X).` against the program of Figure 42. This query produces the subgoals and answers shown in Figure 43. Both subgoals

```
:- table p/1.

p(f(X)) :- p(Y), r(X,Y), ¬q(X).
p(g(X)) :- q(X).
p(Y) :- r(X,Y), ¬p(Y).

q(X) :- r(X,g(X)).

r(a,g(c)).
r(b,g(b)).
```

Figure 42: *A program requiring* DELAYING.

and their answers are shown in chronological order of generation when the scheduling strategy of page 48 is used. Similarly to the previous example, answers of different subgoals

Subgoal	Answer		*Answer_Id*
$p(X)$	`g(b)`	$[\,]$	$p(X)_1$
	`g(c)`	$[\langle \neg \mathtt{p(g(c))} \rangle]$	$p(X)_2$
	`f(a)`	$[\langle p(X), \mathtt{p(g(c))}, p(X)_2 \rangle]$	$p(X)_3$
$p(g(c))$		$[\langle \neg \mathtt{p(g(c))} \rangle]$	$p(g(c))_1$
$p(g(b))$		$[\,]$	$p(g(b))_1$

Figure 43: *The tables created for the query* `?- p(X).`

may be conditional on the same sets of delayed literals (answers $p(X)_2$, and $p(g(c))_1$ in this example). ■

One of the goals of the SLG-WAM is to cleanly integrate SLG and SLD resolution. The introduction of conditional answers gives rise to an extra complication: delay lists must be propagated through the "unfoldings" of predicates that are evaluated using SLD resolution as illustrated by the following example. Section 6.4.2 presents the precise way that this intuitive operation is performed in the context of the SLG-WAM.

Example 6.4.3 We extend the program of Example 6.4.2 to include both a tabled (`pt/1`), and a non-tabled (`pp/1`) alias for predicate `p/1`, along with a predicate `u/1` which produces the unconditional answers of `p/1`. Figure 44 shows the extension, and Figure 45 shows some

```
:- table pt/1, u/1.
pt(X) :- p(X).
pp(X) :- p(X).
u(X) :- pp(X), ¬pt(X).
```

Figure 44: *Addendum to the program of Figure 42.*

Subgoal	*Answer*		*Answer_Id*
$pt(X)$	`g(b)`	$[]$	$pt(X)_1$
	`g(c)`	$[\langle p(X), \mathtt{p(g(c))}, p(X)_2\rangle]$	$pt(X)_2$
	`f(a)`	$[\langle p(X), \mathtt{p(f(a))}, p(X)_3\rangle]$	$pt(X)_3$
$u(X)$	`g(c)`	$[\langle p(X), \mathtt{p(g(c))}, p(X)_2\rangle, \langle\neg\mathtt{pt(g(c))}\rangle]$	$u(X)_1$
	`f(a)`	$[\langle p(X), \mathtt{p(f(a))}, p(X)_3\rangle, \langle\neg\mathtt{pt(f(a))}\rangle]$	$u(X)_2$

Figure 45: *Tables created for the queries* `?- pt(X).` *and* `?- u(X).`

of the tables that are created by the evaluation of queries `pt(X)` and `u(X)`. The answers to `pt(X)` should be self-explanatory. Contrast, however, delay lists of the answers of the `u(X)`. These lists contain delay literals `p(g(c))`, and `p(f(a))` which have been propagated through the non-tabled `pp/1`. Within the context of SLG, SLD evaluation can be seen as dynamic "unfoldings" of predicates within tabled clauses (a precise meaning to this observation is given in [68]). ■

6.4.2 Delay List Representation and Maintenance

The SLG-WAM represents delay lists in the table in a way that SIMPLIFICATION can be efficiently performed on them. However, while performing clause resolution, the current value of each delay list is also temporarily maintained in the heap as a Prolog list, to which delay elements are added in constant time using the method described below. Once an answer is derived, the delay list is copied from the heap, interned, and stored in the answer table (all through the `new_answer` SLG-WAM instruction).

Extending and Interning Delay Lists DELAYING and return of conditional answers extend the delay list of the current generator subgoal by one element which is added to the front of the current delay list in the heap as follows: a new list cell is allocated, its *cdr* is set to point to the old value of the delay list, and its *car* is set to point to a new delay element *elt*. The 3 fields of *elt* are filled by the procedures that implement these operations. Observe that this process of extending the delay lists creates temporary delay lists whose literals appear in reversed order from the literals in the corresponding program clauses. To maintain the correspondence between the original and the residual program, when the answer is about to be stored in the answer table, the elements of its delay list are copied in reversed order. The latter imposes no overhead, since the delay list has to be traversed to be copied from the heap.

Delay List Maintenance The WAM maintains information about the state of the computation in global registers. To keep track of the current delay list, we introduce a new WAM register, named **D** register. Accordingly, all choice point frames used for program clause resolution, are expanded with an extra field (called *Dreg*) that records the value of **D** register at the time of choice point creation. Let S be the nearest tabled ancestor of a subgoal being evaluated. Then **D** register is updated to point to the delay list of the generator subgoal for S, under the following three conditions.

1. When a new tabled subgoal is called, the **D** register is set to point to an empty delay list. This is performed by the SLG-WAM "try" instructions for tabled predicates (tabletry, and tabletrysingle) immediately after the creation of the generator choice point frame (having saved the previous value of **D** register in the generator choice point). **D** register is reset to the empty list by the tableretry, and tabletrust instructions, since they restart program clause resolution for tabled predicates.

2. As is done for all other WAM registers, **D** register is restored to its earlier value in the event of backtracking (through the retry, and trust instructions, as well as the instructions of Section 4.5.3 that backtrack through answer tries). Note that this restoration is unnecessary when backtracking to the generator choice point of a tabled predicate, since immediately afterwards **D** register will be set to point to an empty delay list.

3. Finally, the value of **D** register is also restored in forward continuations of tabled predicates. To illustrate, let

$$p(\overline{a}) \ :- \ l_1(\overline{b_1}), \ldots, l_i(\overline{b_i}), \ldots, l_n(\overline{b_n}).$$

   be the clause under execution, and assume that the subgoal $t_i(\overline{b_i})$ of literal $l_i(\overline{b_i})$ is a tabled subgoal. Observe that, under condition 1, if the call to $t_i(\overline{b_i})$ is new and produces a generator choice point, it will alter the value of **D** register. To compensate for the possible change, upon the success of $t_i(\overline{b_i})$, (for example, through the return of an answer for $t_i(\overline{b_i})$) the value of **D** register is restored to the value saved in the *Dreg* field of the tabled choice point for $t_i(\overline{b_i})$. A similar restoration of **D** register is necessary when resuming a suspended computation. Therefore the restoration of **D** register during forward execution is performed by the new_answer, answer_return, and negation_resume SLG-WAM instructions. Any new delay element (from the return of a conditional answer or success of a negative subgoal with a conditional answer) is then added to the delay list referenced by the restored value of **D** register.

6.4.3 Extending the SLG-WAM Table Space to Represent Conditional Answers

The ability to factor similar operations and avoid useless recomputations has been regarded as fundamental to efficient evaluation of logic programs. An instance of this is the use of tries as the basis of the tabling data structures of XSB. As discussed in [50], tries can often avoid repeated rescanning of common subterms within subgoal and answer tables. However, the tries of [50] allow storage and manipulation of unconditional answers only (i.e. only of the answer substitutions and not of delay lists). As a result, an extension of the SLG-WAM table space is necessary so that delay lists can be stored in answer tables. Basic requirements of this extension are that it should efficiently support SIMPLIFICATION, and it should not impose an overhead on the ANSWER RETURN operation, since delay lists are never copied out of the table. In this section we describe how the tabling data structures of [50] can be naturally extended to meet these requirements.

Recall from Section 4.3.2, that the SLG-WAM associates each tabled subgoal with an answer table containing a set of answers. The set of answer substitutions of these answers are organized in a trie, henceforth referred as the *answer substitution trie*. Its nodes record information about the range of the answer substitutions (using the *atom* field of the nodes). The outgoing transitions from a node are traced using its *child* pointer and by then following the list of *sibling* pointers of this child. Because delay lists represent unevaluated steps of a deduction, an answer substitution may have many delay lists associated with it in a non-stratified program, corresponding to various partial deductions. We therefore factor answer substitutions, associating them with delay lists using the *child* pointer of the leaves of the answer substitution trie. Conditional answer substitutions point to a *DelayInfo* record that, apart from the set of delay lists associated with that answer substitution, also contains information about the positive delay elements of that answer substitution, along with a pointer back to the subgoal. Naturally, leaf nodes of unconditional answer substitutions have null *child* pointers.

Figure 46 provides a close look at the trie data structures of the SLG-WAM by presenting the subgoal trie for predicate `p/1` of Example 6.4.2, and answer tables for two of these subgoals (both these subgoals and their answers are indicated in Figure 47). The fields of the answer substitution trie nodes are shown in the order: *atom, child, parent*, and *sibling*. The use of the *parent* pointer is explained in Section 6.4.4. *DelayInfo* records are shown attached to leaf nodes of conditional answer substitutions.

Recall that upon the derivation of an answer, its delay list is copied from the heap and stored in the answer table. Since, as illustrated by Examples 6.4.1, 6.4.2, and 6.4.3, it is quite common for a particular delay element to appear in more than one delay list, we intern both delay elements and delay lists in global data structures. These are referred to as *Interned Delay Elements* (*IDEs*) and *Interned Delay Lists* (*IDLs*) respectively. Interning elements usually saves space. However, the main advantage of interning is that simplification operations can now be shared: for example, one simplification operation for an interned delay

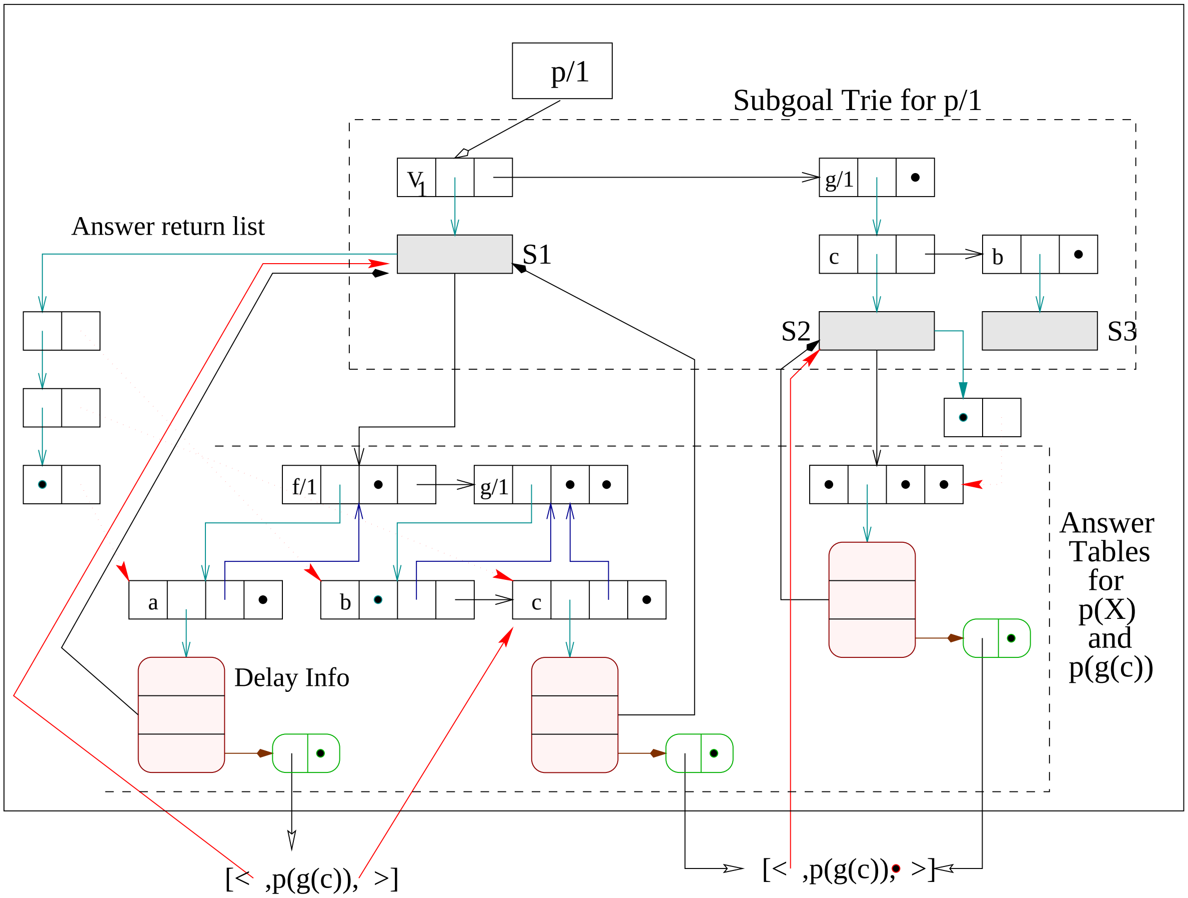


Figure 46: *Subgoal and answer tries for predicate* `p/1` *of Example 6.4.2.*

element *ide* suffices to simplify all delay lists that contain the *ide*. Section 6.5 fully discusses this point.

In SLG, a deduction can be delayed positively on a conditional answer, or negatively on a (ground) negative literal. The structure of IDEs reflect this fact. Positive IDEs are grouped

Subgoals:

$S_1 : p(X)$
$S_2 : p(g(c))$
$S_3 : p(g(b))$

Answers for $p(X)$:

Answer Subst.	*Delay List*
`X = f(a)`	$[\langle p(X), \mathtt{p(g(c))}, p(X)_3 \rangle]$
`X = g(b)`	$[\,]$
`X = g(c)`	$[\langle \neg\mathtt{p(g(c))} \rangle]$

Answers for $p(g(c))$:

Answer Subst.	*Delay List*
	$[\langle \neg\mathtt{p(g(c))} \rangle]$

Figure 47: *Subgoal and answer tables for predicate* `p/1` *of Example 6.4.2.*

together based on their *Answer_Subst_Id* field (which identifies their answer substitution), while negative IDEs are grouped based on their *Subgoal_Id* (which identifies the subgoal of the literal). We can therefore refer to the positive IDEs of an answer substitution, or the negative IDEs of a subgoal without ambiguity.

While the uses of these data structures are detailed at the level of instruction execution in Section 6.5.4, we broadly summarize their uses here. First there is a one-to-many relation between an answer substitution η_S and its (interned) delay lists. It is important to maintain this relation explicitly since the addition of an unconditional answer for η_S means that its conditional answers are unnecessary and should be removed. Next, because the same delay list may occur for many answer substitutions, delay lists are interned as IDLs so that a single simplification operation will suffice for many delay lists (cf. in Example 6.4.2, the delay lists containing $\langle \neg \mathtt{p(g(c))} \rangle$). These IDLs in their turn, consist of delay elements which are interned as IDEs so that they can be grouped together. IDEs contain pointers back to the delay lists that contain them, explicitly maintaining a relation used to propagate simplification (cf. in Example 6.1.1, the propagation of the simplification of $\neg\mathtt{r}$ through the answer for q to the answer for p).

6.4.4 Returning Answers from Answer Tables

Since the generation and the consumption of answers is asynchronous, and new answers can be inserted *anywhere* in the answer table, it is not possible to efficiently perform the ANSWER RETURN operation by sequentially backtracking through the elements of an answer table starting from the root of its answer substitution trie. Consequently, each tabled subgoal, while *incomplete*, is also associated with an explicit list of answers that need to be returned to applicable *active* subgoals. We refer to this list as the *answer return list* (of a subgoal). As mentioned, delay lists are never copied out of the table; hence, the set of answer substitutions of an answer table coincides with the set of answers that have to be returned; i.e. one ANSWER RETURN operation is sufficient for a whole set of answers that share an answer substitution. Furthermore, note that each leaf node of the answer substitution trie uniquely identifies an answer substitution. These two observations suggest that the leaf nodes of the answer substitution trie should be used as elements of the answer return list. Elements of the answer return list appear in chronological order of creation; new answer substitutions are added at the end of the list. For example, in Figure 46 the answer substitutions for $p(X)$ are derived in the order $\mathtt{p(g(b))}$, $\mathtt{p(g(c))}$, and $\mathtt{p(f(a))}$. Answers are returned from answer tables of incomplete subgoals by traversing the answer return list, and, for each of its elements, constructing the answer substitution by a leaf-to-root traversal. To efficiently perform this latter traversal, every node of the answer substitution trie maintains a back pointer to its *parent* node. Later, the answer return list can be reclaimed upon the completion of a subgoal.

Using the leaf nodes of the answer substitution trie as elements of the answer return list has the following indirect advantages:

1. as SIMPLIFICATION does not modify the answer substitution part of the answer, the

elements of the answer return list do not change;

2. the unnecessary traversal of the delay list part of the answer is avoided, thus the cost of the ANSWER RETURN operation is proportional to the size of the answer substitution; and finally

3. since the conditionality of an answer substitution is indicated using a field of the corresponding leaf node of the answer substitution trie, the ANSWER RETURN operation from incomplete tables can immediately determine whether a conditional answer is being returned and, as a result, whether the literal should be added in the delay list of the clause under execution.

Note, however, that answers can be returned from a *completed* answer table by sequentially backtracking from the root of its answer substitution trie. Recall that to efficiently perform the ANSWER RETURN operation from completed tables, answer substitution tries are dynamically compiled into WAM code as described in Section 4.5.3. As the delay list is not traversed during ANSWER RETURN, the majority of the WAM-like instructions that realize this compilation scheme remains unchanged. However, the introduction of conditional answers gives rise to an extra complication: since backtracking is performed starting from the root of the answer substitution trie, it is not known whether a substitution is conditional until the leaf node is reached. The instructions that handle leaf nodes of the answer substitution trie have to therefore be extended to signal the conditionality of the answer substitution, and to appropriately extend the current delay list.

6.5 Simplification

SIMPLIFICATION of delayed literals is necessary so that when subgoals are completed, a ground instance of a subgoal S is true in the well-founded partial model if and only if it is an instance of a unconditional answer, and a subgoal S is false if and only if it has no answers, conditional or otherwise.[2]

6.5.1 Simplification Principles

Example 6.1.1 provides instances of simplification of both positive and negative delayed literals. As mentioned above, not all delayed literals are undefined in the well-founded model of a logic program. Some of the literals that were delayed in order to break negative loops, or to due the return of conditional answers may become known to be true or false later during the process of SLG evaluation. When their truth value does become known, the literal or clause may be simplified away, but until then answers conditional on the delayed literal may

[2]Technically, the truth of the above statement requires the implementation of the SLG ANSWER COMPLETION operation. Our experience has shown that this operation is rarely needed in practical programs, and its omission does not affect the soundness of our results.

be used to derive further answers and may unnecessarily open up the search space. For the efficient evaluation of normal logic programs we therefore adhere to the following principles:

Principle I *Conditions for simplification of delayed literals should be detected, and the* SIMPLIFICATION *operation should be applied, as early as possible.*

Principle II *Derivation of an unconditional answer for a subgoal S should immediately remove from the table for S all conditional answers with the same answer substitution.*

Principle III *Unsupported answer substitutions (see Section 6.1) should be removed from the answer tables as soon as possible.*

6.5.2 Events that Trigger Simplifications

In order to follow Principle I, that is roughly, to simplify as soon as possible, the SLG-WAM performs simplification whenever the truth value of a subgoal or answer becomes known to be true or false. The SIMPLIFICATION operation in the SLG-WAM is driven by the following three events:

1. derivation of an unconditional answer for a subgoal;
2. completion of a subgoal with no answers; and
3. deletion of an unsupported answer substitution.

Derivation of an unconditional answer Occurrence of this event signifies that all instances of the answer will be true in the well-founded model of the program. There are two distinct ways that unconditional answers can be produced: (1) through program or answer clause resolution before the completion of a subgoal; the new answer is then inserted in the answer table by executing a `new_answer` instruction, and (2) through simplification of an existing conditional answer by removing the last delay element from its delay list. Note that in the latter case, the subgoal might have already been marked as completed.
Let A be a new unconditional answer of a subgoal S, and let η_S be its answer substitution. The derivation of A removes all conditional answers with η_S from the system (Principle II), and can trigger simplification of both positive and negative delay elements; (all positive elements of η_S become true, while negative elements of S become false). Note that simplification of positive delay elements takes place only when conditional answers with answer substitution η_S have already been (conditionally) returned to consuming subgoals.

Completion of a subgoal with no answers The subgoal fails. As a result, all negative delay elements of that subgoal now succeed, and can be removed from the delay lists that contain them. At an operational level, a step of the `completion` instruction (cf. Figure 39) checks whether there are any negative IDEs for the failed subgoal, and if so, it triggers their simplification. Note that no direct simplification of positive delay elements can be initiated by a failing subgoal.

Deletion of an unsupported answer substitution Simplification can change the status of an answer substitution η_S for a subgoal S from supported to unsupported when the last conditional answer is removed due to one of its delay elements becoming false. In this case, following Principle III, the engine must address two types of situations. First, all answers delayed (positively) on η_S must be removed from the system. This can cause further simplifications to occur in answers delayed on these conditional answers. Secondly, if S is already marked as completed and η_S was its last answer substitution, then S fails. Simplifications should take place similar to those initiated by completion of a subgoal with no answers.

Note that simplifications tend to have cascading effects. That is, all simplifications can cause (directly or indirectly) simplifications of both positive and negative delayed literals. Having described how SIMPLIFICATION is triggered, we now present the SLG-WAM procedures that perform simplification, and motivate the data structures that support them.

6.5.3 Simplification Procedures

Depending on whether the simplifiable delay element is true or false, the simplification procedures can be partitioned into the following two groups:

Deletion of a delay element from delay lists Simplification procedures of this sort are directly applicable either when a subgoal fails, or when an unconditional answer is derived for a conditional answer substitution. The action that should be performed by simplification procedures of this group is straightforward: all delay lists that contain the simplifiable delay element are shortened by one element. More specifically, when a subgoal S fails all negative delay elements whose $Subgoal\_Id$ is S are removed from the lists where they appear. Similarly, all positive delay elements whose $Answer\_Subst\_Id$ is η_S are removed upon the derivation of an unconditional answer substitution η_S.

Deletion of unsupported answers. Simplification procedures of this category take place when an answer becomes unsupported because one of its delay elements is found to be false. This happens either when an unconditional answer is derived for a subgoal S (in which case all negative delay elements whose $Subgoal\_Id$ is S are false), or when a conditional answer substitution η_S changes status to unsupported (in which case all positive delay elements whose $Answer\_Subst\_Id$ is η_S become false). Note that because of the factoring of answers on their answer substitution, deletion of answers in this case may involve deletion only of delay lists.

In the SLG-WAM, the following four procedures implement the abstract operations specified above:

simplify_neg_fails($Subgoal$) Removes the negative delay elements of a failed subgoal from delay lists.

simplify_pos_unconditional($AnswerSubstitution$) Removes the positive delay elements of an unconditional answer substitution from delay lists.

simplify_neg_succeeds($Subgoal$) Deletes answers that contain negative delay elements of a subgoal that succeeds.

simplify_pos_unsupported($AnswerSubstitution$) Deletes answers that contain positive delay elements of an unsupported answer substitution.

In terms of data structures, the last two simplification procedures cause removal of delay lists and possibly of answer substitutions. To efficiently implement both categories of simplification operations mentioned above, we need information linking each interned delay element *ide* to the set of interned delay lists that contain *ide*, and this information is described in Section 6.5.4. Figures 48 and 49 contain pseudocode for one procedure from each group. The remaining two procedures are dual, and are presented in Appendix B.3. Note that all procedures that implement the simplification operations are mutually recursive. This reflects the fact that simplification operations usually have a cascading effect.

```
Procedure simplify_pos_unconditional(AnswerSubstitution)
 For each PIDE in the IDE list of AnswerSubstitution
   For each IDL in which this PIDE appears
     IDL := IDL − {PIDE}; /* remove the element from the delay list */
     if (IDL = [])              /* answers are now unconditional */
       For each answer A having answer substitution η_S and IDL as delay list
         i. Initiate a simplify_pos_unconditional(η_S) simplification;
        ii. if (the subgoal S containing A now succeeds) then
              Initiate a simplify_neg_succeeds(S) simplification;
```

Figure 48: *Pseudocode for simplifying delay lists by removing elements.*

6.5.4 Data Structures that Support Simplification

As mentioned, the efficiency of the simplification operations heavily depends on the data structures that support them. For example, to efficiently perform simplification triggered by the derivation of an unconditional answer for a conditional answer substitution η_S, η_S should be associated with its delay elements. The interned delay elements need to be associated with the delay lists that contain them as members, and in turn, each of these delay lists needs to be associated with the set of answers that contain them as delay list.

Figure 50 shows the *persistent* data structures that the abstract machine described in this document adds to the WAM, and the relationships between them.[3] Information about

[3]Relationships between elements of Figure 50 do not correspond to any particular program, but illustrate

Procedure simplify_neg_succeeds($Subgoal$)
For each $NIDE$ in the IDE list of the $Subgoal$ /* $NIDE$: Negative IDE */
  For each IDL in which this $NIDE$ appears
    Remove IDL from the IDL Table; /* IDL contains a false literal */
    For each answer substitution η_S such that there is an answer $A = \langle \eta_S; IDL \rangle$
      Remove IDL from the set of delay lists of η_S;
      if (the resulting set is empty) /* the status of η_S is */
        i. Delete η_S; /* now unsupported */
        ii. Initiate a simplify_pos_unsupported(η_S) simplification;
        iii. if (the subgoal S that had η_S as an answer substitution now fails)
          Initiate a simplify_neg_fails(S) simplification;

Figure 49: *Pseudocode for a simplification procedure that deletes delay lists.*

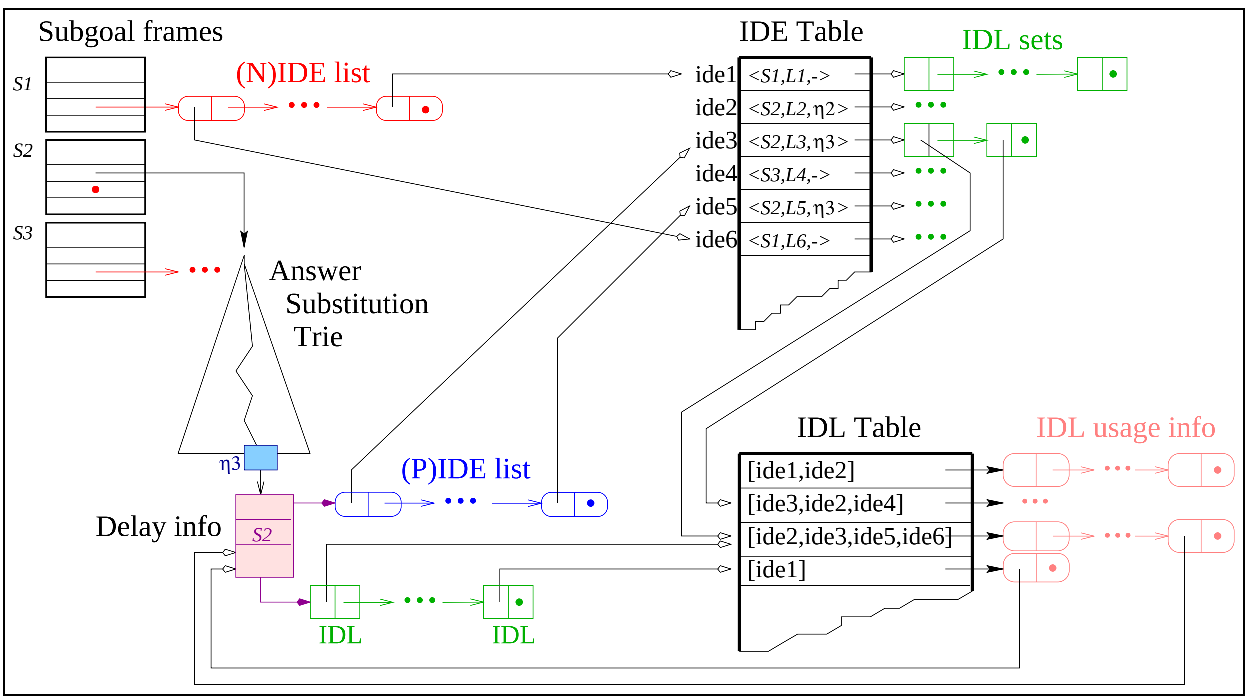


Figure 50: *Relationships between Persistent SLG-WAM Data Structures*

subgoals is recorded in *subgoal frames*; through such a frame, a subgoal S is associated with an answer table (in the form of a trie), and the answer return list of this table. In addition to the information of Section 4.3.2, the subgoal frame of a subgoal S is also associated with the list of negative IDEs of S. The data structures of answer tables were described in Section 6.4.3. However, note the *DelayInfo* record of the conditional answer substitution η_3:

all issues.

it contains information about the positive IDEs of η_3, the subgoal that contains η_3 as an answer substitution, and the set of delay lists associated with η_3.

The delay lists and elements are interned and stored in global tables, the *Interned Delay List (IDL) Table*, and the *Interned Delay Element (IDE) Table*, respectively. The information stored in the entries of these tables is actually in the form of pointers (to subgoal frames, answer substitutions, and entries of the IDE Table); they are not shown as such for readability of the figure. Each entry of the IDE Table is associated with the set of delay lists that contains this entry as an element. Also, each entry of the IDL Table is associated with the conditional answer substitutions that point to this entry.

Everything shown as a list in Figure 50 is actually a set. This includes the IDE Table; its entries, once inserted, appear uniquely. Note, however, that through simplification, the IDL Table can contain duplicates (for example, by simplifying $ide2$ through a simplify_pos_unconditional simplification). Currently, XSB does not collapse these elements. Also, again through simplification, IDLs may become empty (for example failure of $S1$ causes simplification of $ide1$, and as a result, the last entry of the IDL Table becomes empty). Empty IDLs are removed from the table.

6.5.5 Examples of Simplification

Finally, in this section we make concrete how the framework of the SLG-WAM operates by showing how it handles various technical issues arising from simplification. Recall that delay lists of active and suspended clauses are temporarily stored in the heap. As a result, they will not be affected by the simplification procedures which only affect tabled IDE and IDL information. We remedy this by checking delay lists for simplifiability when they are about to be interned and added to the answer table by the new_answer instruction. Then, two conceptually distinct forms of simplification can take place:

1. Removal of some delay elements because they are simplifiable in the current system (even though they were not at the time of DELAYING). Naturally, the simplifiable delay elements do not have to be interned. As an extreme case, a new answer may appear as if it has a non-empty delay list, even though all of its delay elements are currently simplifiable. Example 6.5.1 presents an instance of this situation.

2. Deletion of the entire answer because one of the elements in its delay list fails. In this case the answer is unsupported and need not be added in the answer table, none of the delay elements need be interned, and the computation should fail. An instance of this situation is shown by Example 6.5.2.

Example 6.5.1 Consider the evaluation of a query `?- p.` against the program of Figure 51 where all predicates are evaluated using SLG resolution. The table of Figure 52 presents a chronological listing of the main events that occur while evaluating this query. After detecting two loops through negation, the literal $\neg$`p` in the body of the suspended clause for t gets

```
:- table_all.

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

Figure 51: *Program requiring* SIMPLIFICATION.

delayed. A negative delayed literal ⟨¬p⟩ is added to the front of the delay list of t's clause and is stored on the heap. Execution continues with literal ¬q in the body of t, this literal and the clause of t get suspended, and after several steps the execution of completion instruction detects subgoals p, q, and r forming an unfounded set, and completes them. Since they have no answers, they fail. q's completion resumes the suspended literal ¬q in the body of t; this literal is true, and, since there are no more literals in the body of this clause, the SLG-WAM executes a new_answer instruction for t. Following the rules for updating the **D** register described in Section 6.4.2, the delay list for t is [⟨¬p⟩]. Note however that since subgoal p has been completed with no answers, the delayed element ⟨¬p⟩ is removable from the delay list, and the answer of t is actually unconditional and is added to the table as such. After a simplification step, the computed well-founded model is two-valued: $T = \{t\}, F = \{p, q, r, s\}$. ■

Example 6.5.2 Consider the evaluation of a query `?- r.` against the program of Figure 53. Like in the previous example, all predicates are evaluated using SLG resolution, and the table in Figure 54 presents a listing of the main events that occur while evaluating this query. We again focus the attention on only some of them. Subgoal p gets a conditional answer with delay list [⟨¬q⟩], and gets completed. Its completion resumes the suspended literal ¬p in the body of r, the negative delayed literal ⟨¬p⟩ is added in q's delay list which resides on the heap, and execution continues with literal ¬q that suspends. A negative loop involving subgoals q, r is detected, and the literals ¬r, ¬q in the body of the clauses of these subgoals are delayed. Execution continues in q's body, and since subgoal q is involved in a positive

Negative loop involving p, r; literal ¬r gets delayed
Negative loop involving p, s, t; negative suspensions ¬p, ¬t marked delayed
Delayed literal ⟨¬p⟩ resides on the heap; ¬q, r are called and both suspend
Subgoal s gets a conditional answer s_1 with delay list [⟨¬t⟩]
Answer s_1 returned to s in p's body, q is called
Subgoals $\{p, q, r\}$ get completed and fail; ¬q is resumed and succeeds
Subgoal t is about to get an answer t_1 with delay list [⟨¬p⟩]
However, ⟨¬p⟩ is now known to be true; t_1 is unconditional; t succeeds
Answer s_1 is now unsupported and is deleted by SIMPLIFICATION; s fails

Figure 52: *Outline of the evaluation of* `?- p.`

```
:- table_all.

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

Figure 53: *Program requiring* SIMPLIFICATION.

loop, q gets completed with no answers. This event triggers simplification which produces an unconditional answer for p, so p succeeds. Execution continues with resuming literal $\neg$q in r's body, and an answer with delay list $[\langle\neg\texttt{p}\rangle, \langle\neg\texttt{q}\rangle]$ is derived and is about to be entered in r's answer table. Note however, that since p succeeds, $\langle\neg\texttt{p}\rangle$ is now known to be false, and consequently the new answer is unsupported, and should be ignored. Subgoal r fails. The computed well-founded model is again two-valued: $T = \{p\}, F = \{q, r\}$. ■

Finally, as a very technical point, SIMPLIFICATION can cause deletion and re-insertion of the same answer substitution (especially since many different answers may share the same answer substitution). Example 6.5.3 presents an instance of this situation.

Example 6.5.3 Consider the evaluation of a query `?- r.` against the program of Figure 55 where all three predicates are evaluated using SLG resolution. The table in Figure 56 shows a listing of the main events that occur while evaluating this query, which we, contrary to the previous examples, present in detail. The first invocation of the completion instruction determines subgoals p and q to be an independent SCC where both subgoals depend negatively upon each other, and consequently the first literals in the bodies of the clauses of these subgoals are negatively delayed. As a side comment, note that literal $\neg$p in the body of the first clause of r is not delayed since r is not part of the independent SCC. Literal $\neg$q in the body of p is resumed first. Since that clause contains no more literals, subgoal p gets a conditional answer p_1 with delay list $[\langle\neg\texttt{q}\rangle]$, and p_1 is added in p's answer table. Literal $\neg$p in the body of q gets resumed next, also with a *delayed StratStatus*, a delayed

Negative loop involving p, q; suspensions $\neg$p, $\neg$q marked delayed
In q's clause, delayed literal $\langle\neg\texttt{p}\rangle$ resides on the heap; $\neg$r suspends
Subgoal p gets a conditional answer p_1 with delay list $[\langle\neg\texttt{q}\rangle]$
Subgoal p gets completed; its negation suspensions are resumed
In r's clause, delayed literal $\langle\neg\texttt{p}\rangle$ resides on the heap; $\neg$q suspends
Negative loop involving q, r; suspensions $\neg$r, $\neg$q marked delayed
Execution in q's body; subgoal q gets completed and fails
Simplification of $\langle\neg\texttt{q}\rangle$ produces an unconditional answer for p; p succeeds
Subgoal r is about to get an answer r_1 with delay list $[\langle\neg\texttt{p}\rangle, \langle\neg\texttt{q}\rangle]$
However, $\langle\neg\texttt{p}\rangle$ is now known to be false; r_1 is unsupported and is deleted
Subgoal r fails

Figure 54: *Main events while evaluating the query* `?- r.`

```
:- table_all.

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

Figure 55: *Program requiring* SIMPLIFICATION.

Negative loop involving p, q; suspensions ¬q, ¬p marked delayed
Subgoal p gets a conditional answer p_1 with delay list [⟨¬q⟩]
In q's clause, delayed element ⟨¬p⟩ resides on the heap; ¬r suspends
Negative loop involving p, q, r; suspensions ¬q, ¬p, ¬r marked delayed
In r's 2nd clause, delayed literal ⟨¬q⟩ resides on the heap; ¬r suspends
Subgoal r gets a conditional answer r_1 with delay list [⟨¬p⟩]
In q's clause, delayed literal ⟨¬r⟩ resides on the heap; q is called
Subgoals p, q get completed; q fails and simplification is triggered
Simplification of ⟨¬q⟩ produces an unconditional answer for p; p succeeds
Simplification of ⟨¬p⟩ deletes r_1 (unsupported); r now has no answer
Negative loop involving r; ¬r marked delayed
Subgoal r is about to get an answer r_2 with delay list [⟨¬q⟩, ⟨¬r⟩]
However, ⟨¬q⟩ is known to be true; r_2's delay list gets interned as [⟨¬r⟩]
Subgoal r gets completed with a conditional answer

Figure 56: *Main events while evaluating the query* `?- r.`

element ⟨¬p⟩ will be prepended on q's delay list, and execution in this clause will be again suspended on encountering literal ¬r. Both clauses of subgoal r will also be suspended (on the completion of subgoals p and q respectively), and the second invocation of completion will schedule negation_resume instructions for these suspensions with a *delayed StratStatus*. The delay list of the second clause of r will get a delayed element ⟨¬q⟩, but this clause will once again get suspended, this time on the completion of ¬r. The first clause of r will produce a conditional answer r_1 with delay list [⟨¬p⟩]. Execution continues by resuming ¬r in the body of q with a *delayed StratStatus*, and then q is called. Since there are no unselected literals in clauses that are not suspended, execution fails back to a completion instruction which determines subgoals p and q to be completely evaluated. Subgoal q is completed with no answers, an event which initiates cascading simplifications. A simplify_neg_fails(q) operation is called; this operation removes the single delayed element from the delay list of p_1, and p_1 becomes unconditional. This, in turn, calls a simplify_pos_unconditional(p_1) operation which deletes the now unsupported answer r_1. Since this is the only current answer of r, a simplify_pos_unsupported(r_1) operation is possible, but, as r_1 has not been returned anywhere, this operation has no effect. Since no more SIMPLIFICATIONS are possible, execution continues with a completion instruction that detects a negative loop involving r. The literal ¬r in the body of the second clause of r is marked as delayed, it is added in the delay list of that clause, and execution continues with literal `p` which is true since p has an unconditional answer. Subgoal r is about to get a conditional answer r_2 with delay list $[\langle\neg\mathtt{q}\rangle, \langle\neg\mathtt{r}\rangle]$, but as q is completed with no answers this delay list is simplifiable to $[\langle\neg\mathtt{r}\rangle]$, and r_2 is inserted in r's answer table. Note that the same answer substitution was already inserted in r's answer trie as part of answer r_1, but was later deleted by simplification. The computed well-founded model of this program is three-valued: p is true, q is false, while r is undefined. ■

Chapter 7

Performance

As [69, 61, 68] report, the SLG-WAM adds a relatively small overhead to the SLD evaluation of definite programs (around 15–20%). Furthermore, as we will present in this chapter the additional overhead that allows evaluation of arbitrary normal logic programs according to the well-founded semantics is minimal. On the other hand, as reported in [58, 70], XSB performs significantly better than other deductive database systems of similar functionality for in-memory computation of a wide variety of recursive queries.

At least for range-restricted Datalog programs evaluation methods based on Magic Sets and (variants of) top-down with tabling have been shown by Seki in [63] as asymptotically equivalent. Indeed, the performance numbers of [58, 70] support Seki's result: while the ratios of CORAL vs. XSB or $\mathcal{LDL}$ vs. XSB vary from about 5 times to 20 times faster for XSB over different programs, they tend to be more or less constant as the data size changes. We believe that XSB's performance is entirely due to the efficiency of the SLG-WAM and the mature compilation technology of the WAM upon which the SLG-WAM is based.

Instead of reporting similar performance comparisons of the above systems for programs containing negation, we concentrate on the relative performance of the SLG-WAM with and without the changes needed for evaluation of normal logic programs under the well-founded semantics. We refer the interested reader to [68] for an extensive list of results on the SLG-WAM's performance on definite programs, as well as a more detailed discussion on the reasons of XSB's impressive performance.

A note on the benchmarking environment For XSB, the benchmarking environment was a SPARC 20 with four 50 MHz SuperSPARC processors (although no test used more than one processor), no two-level cache, 128 Mbytes of memory, running Solaris 2.4. A pre-release of XSB version 1.6.0 was used to produce all emulators. The compilation was done with gcc 2.7.0 (using the `-O4` option). For CORAL, the benchmarking environment was a SPARC 2 with 64 MBytes of memory running SunOS 4.1.3. We note that we report no direct performance comparisons of the two systems.

7.1 Overhead for the Evaluation according to the Well-Founded Semantics

In order to measure the overhead incurred for the increased functionality for computing answers according to the well-founded semantics, we compared the performance of the abstract machine for the evaluation of normal logic programs with that used for the evaluation of definite and fixed-order dynamically stratified programs. We first measured the cost of our additions on standard Prolog benchmarks from the D.H.D. Warren testsuite (see [76] for a description of the programs) using the following types of emulators:

XSB-def Performs SLG evaluation for definite programs only. This emulator does not need the **PT** register and **D** register (and their associated fields in choice points), all answers are unconditional, and completes subgoals by approximate SCC detection (as described in Chapter 4).

XSB-lrd Performs SLG evaluation for fixed-order dynamically stratified programs (as described in Chapter 5). The **D** register is not needed, and all answers are unconditional.

XSB-wfs Performs full SLG, evaluating normal logic programs according to the well-founded semantics. It contains all features of the SLG-WAM described in this document.

	`deriv`	`nreverse`	`qsort`	`query`	`serialise`
XSB-def	1	1	1	1	1
XSB-lrd	1.009	1.001	1.011	1.018	1.008
XSB-wfs	1.019	1.008	1.018	1.033	1.018

Table 1: *Normalized times for executing standard Prolog benchmarks in XSB.*

As can be seen from Table 1 the overhead of the SLG-WAM changes for computing according to the WFS to SLD resolution is negligible (usually less than 2%). These times show the effect of adding two registers to the SLG-WAM for definite programs, and having two extra words in all choice point frames.

Next, we measured the overhead of the extended functionality on typical benchmark programs that use definite SLG resolution. Table 2 contains normalized times of executing a series of queries against a right-recursive transitive closure (over a chain, a cycle, and a full binary tree data structure; all of size 8k), and a same generation program (over a randomly generated 24x24x2 cylinder [58]). None of these programs has negative literals. This set of benchmarks measures the overhead of extra WAM registers, having more words in choice points and completion stack frames, performing *early completion*, checking whether answers are conditionally returned and whether exact SCC detection should be used for the

	TC-chain	TC-cycle	TC-tree	same generation
XSB-def	1	1	1	1
XSB-mod	1.008	1.009	1.012	1.010
XSB-lrd	0.947	0.945	0.979	0.961
XSB-wfs	1.010	1.011	1.021	1

Table 2: *Normalized times for executing standard tabling benchmarks in XSB.*

completion of subgoals (functionality which is not needed for these programs). The times in Table 2 indicate that the overhead of these operations to tabled evaluation of definite programs is minimal (around 1-2%). The speed-up due to the early completion operation (and the simplified completion instruction when no negative suspensions are present), which is necessary for the evaluation of programs containing negation, balances the overhead of computing answers according to the Well-Founded Semantics. Note however, that even without taking this speed-up into account this overhead is still very small; (the ***XSB-mod*** emulator is identical to ***XSB-lrd*** but does not perform early completion). It can be proven (see [68]) that ***XSB-mod*** correctly evaluates the class of (left-to-right) modularly stratified programs.

7.2 Overhead of Other Evaluation Strategies for Stratified Negation

To put the numbers of the previous section in perspective, we also measured the overhead of other implemented evaluation strategies for stratified negation proposed in the literature. One of the more influential of such strategies is Ordered Search [52], a strategy implemented in the CORAL system [53], which evaluates programs that are (left-to-right) modularly stratified.

Tables 3, and 4 compare the performance of Ordered Search with the default evaluation strategy of CORAL: *Seminaive fixpoint evaluation with supplementary Magic rewriting.* We note that Ordered Search is the default strategy of CORAL only for programs that contain negation or multiset grouping.

	TC-chain	TC-cycle	same generation
Semi-Naive	1	1	1
Ordered Search	1.42	1.45	1.30

Table 3: *Performance of CORAL's evaluation strategies on definite programs.*

CORAL provides many annotations that affect the performance of programs; for both evaluation strategies the timings reported are the best that could be obtained by setting these options. The results in Table 3 show that, on definite programs same as those of Table 2, Ordered Search is considerably less efficient than ordinary Seminaive fixpoint evaluation (around 40% slower).

We also measured the performance of Ordered Search against Seminaive evaluation on programs containing negation. In general, programs containing negation require evaluation using Ordered Search; however, in cases where the negation is stratified and no positive loops are encountered, negation can be safely combined with ordinary Seminaive or CORAL's *pipelined* evaluation (in similar ways that negation works in many programs evaluated using SLDNF). The benchmarks `even`, and `congested` (taken from the CORAL examples directory) are simple examples of such programs: they contain stratified uses of negation, and, in `congested`, a left-recursive transitive closure is applied on an acyclic data structure.

	`even`	`congested`
Semi-Naive	1	1
Ordered Search	1.71	12.33

Table 4: *Performance of CORAL's evaluation strategies on programs with negation.*

As can be seen from the results of Table 4, Ordered Search imposes an unnecessarily big performance penalty (for `congested` over an order of magnitude slowdown) on the execution of programs that do not need its power. Contrast the CORAL numbers with the performance of tabled (`tnot/1`) vs. Prolog-style (`not/1`) negation shown in the last two rows of Table 5. We also note that the first row of the same table represents the performance of the program where SLDNF evaluation is used (no predicate is tabled) — in this case, the left-transitive closure of `congested` was hand-transformed to a right-recursive one.

	`even`	`congested`
SLD-`not/1`	0.72	0.71
SLG-`not/1`	1	1
SLG-`tnot/1`	1.19	1.23

Table 5: *Performance of different types of negation in XSB.*

Besides its poor performance characteristics, it should also be mentioned that according to the CORAL manual, Ordered Search is restrictive (at least in its current implementation); for example it prohibits the use of certain magic set optimizations like *NRSU factoring* [41] or *SCC-by-SCC* evaluation. More importantly, as the manual indicates, Ordered Search is heavily affected by the number of rules in a module. In contrast, our evaluation method for well-founded negation does not suffer from such symptoms.

These performance numbers indicate a small overhead for the additional functionality of well-founded negation compared to Ordered Search. We believe that these results reflect fundamental aspects of the computation strategies involved, rather than accidents of implementation. Part of the overhead of Ordered Search lies in its maintenance of a set of *context nodes* which together serve as an analogue to the subgoal dependency graph. While, as Section 5.3 indicates, it is a simple matter to use the SLG search tree (as implicitly present in the choice point stack of the SLG-WAM) to maintain dependencies between subgoals, such a structure does not naturally follow from a Seminaive evaluation such as CORAL's. Rather, it must be built from scratch, leading to the observed overheads. The SLG-WAM's small overhead is especially striking since, as mentioned, XSB has been shown to be about an order of magnitude faster than CORAL for definite Datalog queries.

7.3 Performance of Well-Founded Negation

To our knowledge, the `win/1` program is still the only standard benchmark in the well-founded semantics literature. Its SLG and SLDNF versions, using XSB, are shown below. SLDNF resolution is sufficient for acyclic `move/2` graphs (like chains and trees). In this cases the well-founded model is total.

SLG	`:- table win/1.` `win(X) :- move(X,Y), tnot(win(Y)).`
SLDNF	`win(X) :- move(X,Y), not(win(Y)).`

Figure 57 contains timing results (in seconds) for executing the query `?- win(1),fail.` over different data structures of sizes varying from 16 to 8K. Times shown are for 50 iterations. The tabled evaluation of this query creates many distinct subgoals and suspended nodes. For chains and trees, the program is modularly stratified and each subgoal either has one answer, or fails. Figure 57 shows that for the case of the chain, the performance of tabled negation is comparable to that of the negation of Prolog (less than 2 times slower). This difference is due to the recording of subgoals and answers in tables, of suspending computations and of performing completion. In the case of the complete binary tree, however, tabled negation is much slower. This is because tabled negation has to fully evaluate negative literals before completing them, and consequently, it traverses the entire search space which is exponential in the height of the tree (see [12, 58]).

The SLD version of `win/1` will not terminate if the base relation `move/2` contains cycles. Under the Well-Founded Semantics, all answers of `win/1` over a cyclic data structure are *undefined*. The evaluation of the tabled version of `win/1` over a cyclic data structure requires the negative loop checking mechanism described in Section 6.2, and exact SCC detection to decide whether DELAYING is needed. All answers are conditional containing one element

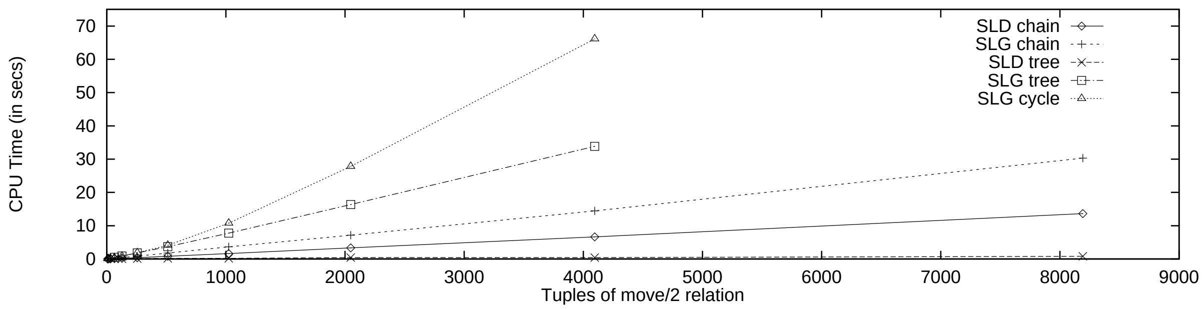


Figure 57: *Performance of the* `win/1` *program over different data structures.*

in their delay list; these elements and delay lists are interned and stored in global tables. Since DELAYING is always needed and no SIMPLIFICATION is possible, performing exact SCC detection and building up data structures to support simplification, imposes an unnecessary overhead to the evaluation of this program. We hold, however, that avoiding DELAYING, providing an evaluation strategy close to the ideal measure of relevance, and performing SIMPLIFICATION as early as possible is crucial for the (search-) efficient evaluation of practical normal programs using SLG resolution.

Providing a set of benchmark problems that will enable systematic experimental evaluation of different resolution strategies and systems that compute the well-founded semantics is still an open issue. *TheoryBase* [15], a tool to create default theories and logic programs for experimentation with nonmomotonic reasoning systems is an interesting recent step in that direction. Since further experimental evaluation of the characteristics of the SLG-WAM when it comes to non-stratified programs is needed, it is in our future plans to use this system to provide a performance evaluation of well-founded negation over a wider (and hopefully more representative) range of problems.

Chapter 8

Related Work

Related work concerning the introduction of negation in logic-based languages, stratification classes, and semantics of normal logic programs has been already presented in Chapter 2. In this chapter, we first compare SLG_{RD} with other evaluation strategies for computation under the well-founded semantics, and we then review the main implementation efforts.

8.1 Evaluation Strategies for Computing the Well-Founded Semantics

Earlier work focussed in extending known evaluation strategies for definite programs to the class of stratified Datalog programs. Seki and Itoh [64] formalize OLDTNF by augmenting OLDT [71] with the negation as failure rule; special care is taken so that negative subgoals of lower strata are completely evaluated before their use in higher strata. Kemp and Topor [31] present an extension of Vielle's QSQR/SLD-AL query evaluation procedure for definite queries and databases [77], to a procedure called QSQR/SLS for normal queries and stratified databases. The new procedure is proved sound and complete by induction on the level (stratum) of an allowed normal query. However, both the description and the proofs are restricted to function-free databases. The authors admit that this restriction may be difficult to remove and the problem of finding a complete generalization of QSQR/SLS for programs with function symbols remains open. Ross [56] further generalized QSQR/SLS resolution to the class of modularly stratified programs. In the same paper, he also proposed an extension of a bottom-up evaluation based on magic-sets by incorporating at *run-time* information about negative dependencies through a *depends* meta-predicate. Facts representing transitive dependencies among subgoals were computed explicitly. However, mechanisms for efficient maintenance and sharing of mutual subgoal dependencies were not explored. Both OLDTNF and QSQR/SLS may perform repeated computations when negative contexts are associated with subgoals. Furthermore, none of these papers addresses issues related to the implementation of the proposed strategies.

Bidoit and Legay [7] presented *Well!*, an evaluation procedure for the well-founded semantics that applies to all non-floundering programs whether stratified or not. The core of *Well!* is SLD-AL resolution of Vielle [77] with its three main features: *admissibility test*, *lemma resolution* (which roughly correspond to the SLG's operations for checking for the existence of the subgoal and answer resolution, respectively), and *iteration*. In addition, a test for negative loop checking, *admissibility test for negative subgoals*, and an operation for resolving *potential lemmas* (subgoals with potentially true answers — subgoals with an associated negative context) are introduced. While the resulting evaluation strategy is effective, the use of negative contexts prevents the full sharing of answers across different negative contexts. In the worst case, a subgoal may be evaluated in an exponential number of distinct negative contexts.

Stuckey and Sudarshan in [66] extend the Ordered Search technique to handle well-founded negation. Their extension, called Well-Founded Ordered Search (WF-OS), supports goal-directed query evaluation and, like SLG, allows an arbitrary computation rule. Described simply, WF-OS uses Ordered Search [52] to find dependencies between subgoals, and when a cyclic negative dependency is found, WF-OS applies the (costly) alternating fixpoint technique of Van Gelder [74] to compute the well-founded model of only the set of subgoals that are involved in the cycle. As a special case of the above, for modularly stratified programs WF-OS reduces to Ordered Search and performs no irrelevant computation. There are three interesting differences between WF-OS and SLG_{RD}, our restriction of SLG resolution. The first is that WF-OS does not contain a mechanism like early completion; a mechanism that usually eliminates many spurious negative cycles, consequently reducing the need for dynamic search, and effectively allowing the evaluation of LRD-stratified programs without any expansion of the search space through unnecessary uses of DELAYING. Consequently, SLG_{RD}'s search space is at least as small as that of WF-OS. Secondly, WF-OS eagerly constructs the subgoal dependency graph upon entering a negative context, while in SLG_{RD} the exact dependency graph is constructed lazily whenever an approximation of a strongly connected component AC contains negative dependencies within AC (cf. Section 5.6). Furthermore, this approximation and the *early completion* of subgoals restrict the construction to a relatively small subset of the dependency graph of the program/query pair. Finally, when SLG_{RD} delays a negative literal, all its negative dependencies that caused the use of DELAYING are removed (cf. Section 6.2.1), This operation in effect moves the literal back in the sip order and incrementally reduces the number of negative cycles. Thus, SLG_{RD} is able to relate positive cycles in unfounded sets directly to positive cycles in the subgoals of the completion stack. In contrast, WF-OS cannot make this connection since it does not update its dependency information when it introduces $un(\neg p(\overline{a}))$ facts (WF-OS's equivalent to DELAYING).

Based on the concept of *search forest* of [9] whose goal is to separate the search space from the search strategy, Bol and Degerstedt in [8] define a top-down tabulated version of SLS resolution and proved its soundness and completeness w.r.t. the well-founded semantics.

In contrast to SLS resolution as proposed by Przymusinski and by Ross, the severe restriction of a positivistic computation rule is lifted, loops are detected by tabulation, and the resulting resolution method is effective. Interestingly, they also define a notion of *waiting* and *blocked* subgoals that roughly corresponds to SLG_{RD}'s *suspended* status of X-rules (cf. Definition 3.3.6). However, despite the fact that only one forest for each negative context is constructed, the propagation of non-ground negative literals may result in an exponential number of distinct conditional answers for a subgoal. Thus, there is at least a question as to whether the resolution method of [8] preserves the polynomial data complexity of the well-founded semantics.

Using results from [8], Degerstedt and Nilsson in [20] introduce a new magic templates transformation and give a new fixpoint characterization of the well-founded semantics. The later is then used to show a step-by-step correspondence between the naive bottom-up evaluation of the transformed program and a class of top-down search strategies defined in terms of the *search forest* of [9]. In effect, they provide an upper bound on the search space that must be considered in order to preserve completeness of the bottom-up/magic templates approach. The problem of actually computing answers to queries in a search-efficient manner is not addressed.

8.2 Well-Founded Semantics Implementations

The SLG-WAM is not the first implementation of SLG resolution. After initial experimentation with XOLDTNF, Chen and Warren formalized SLG and developed a meta-interpreter for it on top of Quintus Prolog (the system was later ported to Sicstus Prolog and XSB). To perform COMPLETION, the SLG meta-interpreter uses a simple subgoal dependency maintenance algorithm that over-approximates strongly connected components (see [12]). As a result, the meta-interpreter does not tightly control the use of the DELAYING transformation and thus may unnecessarily expand subgoals that are irrelevant to the query (even in programs that are left-to-right modularly stratified). A certain level of integration of SLG and SLDNF (Prolog) execution is achieved for individual predicates, but due to the naturally limited control of the meta-interpretation, this integration is not as unlimited as that achieved by the SLG-WAM. However, the resulting system was successfully used as a testbed for experimenting with the addition of several different non-monotonic reasoning capabilities to Prolog programs. Specifically, besides query evaluation under the well-founded semantics, extensions for stable model semantics [13] and meta-predicates for abductive and skeptical reasoning based on stable models were recently added to the meta-interpreter [10].

Morishita [40] modified the alternating fixpoint of [74] so that certain problems related to the bottom-up evaluation of the well-founded model of a magic-transformed program were alleviated. Specifically, it was soon noticed that the well-founded model of a "magic" program may not agree with that of the original program. Morishita proposed a solution that generated fewer magic facts in comparison with other published methods, effectively restricting the number of irrelevant subgoals that are encountered in a magic-set evaluation of a program with negation. No minimality results were given (indeed, the resulting alternating fixpoint could perform much worse than the "naive" algorithm even on modularly stratified programs), and the work was restricted to function-free programs. However, the algorithm was implemented as part of the Glue-Nail database system. As reported in [12], the SLG meta-interpreter running under Quintus Prolog performed competitively to Morishita's method, and in programs containing stratified negation both systems were between 30 to 100 times slower than an earlier version of XSB.

Finally, very recently Niemelä and Simons [42] described an implementation of semantics for normal logic programs, with an emphasis on stable model computation. Their implementation is restricted to range-restricted Datalog programs with negation, and the evaluation is completely bottom-up without any goal-orientation. The computation of stable models works in two phases. In the first, a subset of the ground instantiation of the program is produced using an algorithm that guarantees that no stable models are lost due to insufficient grounding. Then, stable models are generated based on bottom-up backtracking search and a pruning method employing an approximation technique for stable models which is closely related to the well-founded semantics. As extra bonus, their implementation includes an efficient (quadratic time) algorithm for computing the well-founded model of ground programs. The latter is based on a modified version of the known linear time algorithm for testing the satisfiability of propositional Horn clauses of Dowling and Gallier [24]. Overall, in tests involving stable model computation of ground programs, their implementation compares favorably with the efficiency of the SLG meta-interpreter.

Clearly the goals of our work and theirs are completely different; interestingly however, they seem to fit well together. Recall that to answer a query Q against a program P according to the well-founded semantics, SLG_{RD} evaluation constructs a final system consisting of a set of answer clauses. This set of answers can be viewed as a *residual program* of the program-query pair $\langle P, Q \rangle$, and consists of all answers to the query plus the rules of all head atoms upon which the answers of the query depend through delayed literals in answer bodies. Indeed, the SLG-WAM contains an explicit representation of this residual program in its subgoal and answer tries and in its IDE and IDL tables (see Section 6.5.4). Furthermore, this residual program (which in most cases is ground) can be further processed according to other forms of non-monotonic reasoning. It seems natural to use the efficient mechanisms of the SLG-WAM as a preprocessor to a system like that of Niemelä and Simons, thereby adding goal-orientation to such a system, efficient handling of non-ground programs, and more importantly restricting the stable model computation to those atoms that are undefined in the well-founded model. The theoretical and practical issues related to the (efficient) implementation of such a system is an interesting topic of future research.

Chapter 9

Discussion

The last two decades have witnessed the development and application of numerous logic programming systems based on Prolog. While the flexibility and performance of these systems demonstrates both the power and practicality of logic-based approaches, deficiencies in Prolog's evaluation strategy have limited the full exploitation of the logic programming paradigm. For instance, SLD's susceptibility to infinite loops and redundant subcomputations prevented Prolog from being used as a query language for deductive databases. In addition, SLDNF's treatment of negation rendered Prolog inadequate for non-monotonic reasoning applications that need a higher-level of expressiveness.

The tabled-based evaluation of normal logic programs addresses these limitations. As a result, the use of tabling has the potential of radically expanding the practical uses of logic programming. Indeed, tabled-based evaluation is beginning to attract the interest of (at least) the research community and a number of application areas that were beyond the reach of Prolog-based logic programming systems have now been made feasible. For example, it has been shown in [19] that simple Horn representations of logic and functional program analysis methods, as executed by the SLG-WAM, can compete with or even surpass, special-purpose program analysis routines written in C. Furthermore, initial experiments indicate that, under the reduction to Horn clauses specified in [82], the SLG-WAM can efficiently perform automated verification of correctness of program properties.

These applications, while exciting, mainly concern definite programs. We believe that the recent expansion of the SLG-WAM to compute the well-founded semantics will serve as a computational basis for even more applications in areas requiring higher declarativity, such as deductive databases and non-monotonic reasoning. For instance, the well-founded semantics (perhaps expanded to include explicit negation in addition to the usual default negation) has been shown to be expressive for reasoning using non-monotonic inheritance [32], for diagnosis and reasoning about actions [43], and as underpinnings for the behavior of collaborative agents. The practical implementation of applications such as these forms the next avenue of research for XSB and for tabling in general. We hope that this thesis will provide a conceptual model for a new generation of systems based on ideas from logic programming.

Appendix A

Proofs from Chapter 3

A.1 Lengthy Proofs

Proof of Proposition 3.3.1: (a) Let Π_P be the maximum number of literals in the body of a rule in P. Let $\mathcal{S}$ be any system in an SLG_{strat} derivation for Q, and $(A : \rho)$ be any pair in $\mathcal{S}$, where A is a subgoal and ρ is its X-sequence. The length of ρ is bounded based upon the following observations:

- When a subgoal A is added to a system, its initial X-sequence contains the X-rules that are obtained by resolving $A \leftarrow A$ on A in the body with rules in P. The number of literals in the body of each X-rule is bounded by Π_P. Since P is finite, the initial X-sequence of A is finite.

- When a transformation extends the X-sequence of A, X-elements are appended to the end of the X-sequence. Let α be an ordinal and let G be the X-rule of a subgoal A corresponding to α. We discuss the possible forms of X-elements e that are appended.

  (i) If e is $disposed\langle\alpha\rangle$, then G is disposed and can be disposed at most once by the construction of an SLG_{strat} derivation;

  (ii) If e is $suspended\langle\alpha\rangle$, then G is suspended on its selected literal. Since the NEGATION SUSPENSION transformation requires that G be an active X-rule and that the positive counterpart of the selected literal be evolving, G can be suspended at most once on each of its negative literals;

  (iii) If e is $unsuspended\langle\alpha\rangle$, then G is marked as unsuspended and, since each atom is completed at most once (through the COMPLETION transformation), a suspended X-rule can be unsuspended at most once for each of its negative literals;

  (iv) If e is $G'\langle\alpha\rangle$, where G' is an X-rule, then G is disposed and replaced by G' (using the NEGATIVE RETURN transformation);

  (v) If e is $G'\langle\alpha, H\rangle$, where G' is an X-rule and H is an atom, then G' is obtained from G by solving the selected atom of G with an answer that has H in the head, and G' has one literal less than G. The number of such X-rules G' that can be obtained from G by POSITIVE RETURN is bounded by the number of distinct atoms (that are not variants of each other), which is countable.

Therefore for each X-rule G of a subgoal A corresponding to an ordinal α, the number of X-rules G' that can be obtained directly from G is bounded by the number of distinct atoms, which is countable. In both (iv) and (v), the number of literals in the body of G' is one fewer than that in G. For any chain of X-rules, $G_0, G_1, \ldots, G_l$, where each $G_{j+1} (0 \leq j < l)$ is obtained from G_j by some transformation, $l \leq 3\Pi_P$; the longest chain is created when each of the literals in the body of G_0 is a negative literal that gets suspended, unsuspended, and deleted through a NEGATIVE RETURN. Thus, there exists some countable ordinal λ by which the length of the X-sequence ρ of A in $\mathcal{S}$ is bounded. The ordinal λ depends upon only the finite program P and the language $\mathcal{LF}$ that is countable, and is applicable to the X-sequence ρ of any subgoal A in any system $\mathcal{S}$ in an arbitrary SLG_{strat} derivation for Q.

For (b), let $\mathcal{S}_0, \mathcal{S}_1, \ldots, \mathcal{S}_\alpha$ be an SLG_{strat} derivation for Q. By definition, $\mathcal{S}_0$ is the empty system $\emptyset$. For each successor ordinal $i+1 (0 < i+1 \leq \alpha)$, $\mathcal{S}_i \sqsubseteq \mathcal{S}_{i+1}$ since each transformation either adds a new subgoal to $\mathcal{S}_i$ or extends the X-sequences of some subgoals in $\mathcal{S}_i$. For a limit ordinal $i (0 < i \leq \alpha)$, $(A : \rho) \in \mathcal{S}_i$ if and only if A is a subgoal in $\mathcal{S}_j$ for some $j < i$ and $\rho = \bigcup\{\rho' | (A : \rho') \in \mathcal{S}_j \text{ and } j < i\}$. By (a), there exists some ordinal λ by which the length of ρ' for $(A : \rho') \in \mathcal{S}_j$ for each $j < i$ is bounded. Thus $\bigcup\{\rho' | (A : \rho') \in \mathcal{S}_j \text{ and } j < i\}$ is well defined. Clearly $\mathcal{S}_j \sqsubseteq \mathcal{S}_i$ for every $j < i$. This concludes the inductive proof of (b).

For (c), since the number of distinct subgoals (that are not variants of each other) is countable and the length of the X-sequence of a subgoal in a system is bounded, the size of a system is also bounded. As each transformation increases the size of a system, and each SLG_{strat} derivation for Q is a monotonically increasing sequence of systems, there must exist some SLG_{strat} derivation for Q that ends with a final system. $\diamondsuit$

Proof of Theorem 3.3.3: Since $\mathcal{S}$ is flummoxed, by Definition 3.3.12 there exists a set Σ of subgoals in $\mathcal{S}$ that is not completed in $\mathcal{S}$. Furthermore, since $\mathcal{S}$ is a flummoxed system, no transformation is applicable to $\mathcal{S}$; in particular, the COMPLETION transformation is not applicable to any set of subgoals Σ' from Σ. Therefore, for every $\Sigma' \subseteq \Sigma$, Σ' is not completely evaluated. First, note that no subgoal $A \in \Sigma$ is completely evaluated, or else the set $\{A\}$ could be completed through a COMPLETION transformation. Furthermore, A does not succeed and contains at least one X-rule that has a selected literal and is not disposed, i.e. it contains at least one *suspended* or *active* X-rule. Then,

- If A contains at least one suspended X-rule G, then its selected literal L is of the form $\neg B$ where B is a subgoal in $\mathcal{S}$, so A depends negatively on B. Furthermore, B cannot be completely evaluated since in that case a NEGATIVE RETURN transformation would be applicable contradicting the assumption that $\mathcal{S}$ is a flummoxed system.

- If A contains only active X-rules, then note that it contains at least one X-rule whose selected subgoal $L = B$ is not completely evaluated (i.e. $B \in \Sigma$).

$\diamondsuit$

Proof of Lemma 3.4.1: First, note that since P and Q are ground, the subgoals that are contained in systems of any SLG_{strat} derivation for Q are also ground. The proof will be by induction on the dynamic stratum of the ground subgoal A.

- Base Case: The dynamic stratum of A is 1.

  Suppose $A \in T_S$. In this case, A is derivable via clauses containing no negation in their bodies and there was a sequence of NEW SUBGOAL, COMPLETION, and possibly POSITIVE RETURN transformations used to derive A. In this case, that $A \in T_{J_1}$ follows directly from the correctness of SLG resolution.

  Next, suppose $A \in F_S$. In this case either there are no clauses whose head is A, or all clauses for A contain a *failing prefix* so that A is derived directly through only NEW SUBGOAL and COMPLETION transformations. Again, the fact that $A \in F_{J_1}$ is immediate.

- For the inductive case, assume that the statement is true for all subgoals whose dynamic stratum is less than N to prove that it is true for a subgoal A whose dynamic stratum is N.

  First suppose $A \in T_S$. In this case the derivation uses the same transformations — NEW SUBGOAL, POSITIVE RETURN, and COMPLETION — as in the base case, and the fact that $A \in T_{J_N}$ follows directly from the soundness of SLG.

  Next suppose A is a subgoal which is an element of the set A_{set} of subgoals in F_S and whose dynamic stratum is N. To prove the statement we consider induction on the order of completion for the goal A' in A_{set} within $\mathcal{E}$.

  - Base case: A' is completed in the first COMPLETION operation for goals in A_{set}. Now A_{set} may consist of several SCCs which are completed at the same time. To demonstrate that the statement is true for all subgoals in A_{set} an induction must be done on the SDG for $\mathcal{S}$. (Such an induction is straightforward and we do not provide it here). Instead we assume that A' is in the lowest of these SCCs. Choose some clause C whose head is A', and consider the actions of the SLG_{strat} derivation. First, a possibly empty sequence of literals are resolved away from the body of C until some L_i is encountered which fails. There are three cases in which L_i may fail. In the first two, the subgoal S_i corresponding to L_i is completed before A'. (1) Suppose L_i is negative. Then S_i must have a lower dynamic stratum than A' and the failure of C is sound by the outer induction assumption. Next assume L_i is positive. In case (2), S_i is of a lower dynamic stratum than A, so that the failure of C is sound due to the outer induction hypothesis. For the final case, (3) assume L_i is positive and S_i is completed by the same COMPLETION operation as that for A. In this case, at the time of completion for A no goal in the SCC of A can depend negatively on any other goal, since this would contradict the assumption that these goals are completely evaluated. Given this property, it is straightforward to show that the subgoals in the SCC will be in F_{J_N}. To see this note that they must be in $F_{J_N}^{\downarrow 0}$ by its definition, and that every clause for each goal must either belong to one of the first two cases, corresponding to the condition that the L_I is false in J_{N-1} or to this third case. For clauses that belong to the third case, note that the goals presence in F will ensure their presence in $\mathcal{F}_I(F)$.

– Inductive case: assume the statement is true for all subgoals of dynamic stratum N completed before A. The only difference from the (inner) base case occurs in case (2) when the failing literal L_i is positive. In this case its corresponding subgoal S_i may either be in a lower dynamic stratum than A or may be in the same dynamic stratum as A but completed earlier. Thus the inclusion of the completed subgoals in F_{J_N} will hold either for the same reasons given in the base case, or because of the inner induction hypothesis.

$\diamondsuit$

Proof of Lemma 3.4.2: (Sketch) Suppose $\mathcal{S}_\alpha$ is not completed. Then by Lemma 3.3.2, $\mathcal{S}_\alpha$ must be flummoxed. of an SLG_{strat} evaluation $\mathcal{E}$ of Q, and $U_\mathcal{S}$ be the set of uncompleted subgoals in $\mathcal{S}_\alpha$, and let N be the minimal dynamic stratum of any element in $U_{\mathcal{S}_\alpha}$. The proof uses induction on N to show that the minimal dynamic stratum for $U_{\mathcal{S}_\alpha}$ cannot occur for any N, which in turn will show that $\mathcal{S}_\alpha$ cannot be a flummoxed system.

- Base case, $N = 1$. In this case F_1 consists only of subgoals that do not unify with program clauses. Such subgoals will be completely evaluated as soon as they are called, and so cannot be uncompleted in a flummoxed state. T_1 consists of atoms that can be proven using clauses that require no negation. It is an easy matter to see that each of these elements must contain an answer when they are completely evaluated and so cannot be a part of a flummoxed state.

- Inductive case: assume that the statement holds for all dynamic strata less than N to show that the statements also holds at stratum N. We prove this case by contradiction: assume $Subgoal$ is an element of $U_\mathcal{S}$ whose dynamic stratum is N. To see that $Subgoal$ cannot be an element of T_{J_N} consider that by Definition 3.2.5 if $Subgoal$ has dynamic stratum N, then it has a derivation consisting of subgoals with strata $N-1$ or less, possibly along with subgoals whose dynamic stratum is N and that are in T_{J_N}. By the induction hypothesis any subgoals having dynamic strata less than N that $Subgoal$ depends on must be completed, and by Lemma 3.4.1 they must be completed correctly. This leaves only subgoals in T_{J_N} that $Subgoal$ may depend on. At this point let $U_\mathcal{S}^{T_{J_N}}$ be those elements of $U_\mathcal{S}$ which are in T_{J_N}. Then one of these elements has a derivation which depends on no other element in $U_\mathcal{S}^{T_{J_N}}$ for it to be in T_{J_N} (otherwise $U_\mathcal{S}^{T_{J_N}}$ would be unfounded). However in this case, that element is completely evaluated and cannot be in $U_\mathcal{S}$. Thus $U_\mathcal{S}^{T_{J_N}}$ must be empty and $Subgoal$ cannot be in T_{J_N}.

Next, we show that no element of $U_{\mathcal{S}}$ can be in F_{J_N}. Consider that each element in $U_{\mathcal{S}}$ must have in its dependency graph a subgoal with a suspended clause — otherwise some subset of $U_{\mathcal{S}}$ would be completely evaluated, contradicting the assumption. Consider now the element $Subgoal$ with associated suspended clause C. $Subgoal$ is suspended on another goal $Subgoal_{neg}$ in $U_{\mathcal{S}}$ so that there is a negative dependency from $Subgoal$ to $Subgoal_{neg}$ in $U_{\mathcal{S}}$, and $Subgoal_{neg}$ is the selected literal for C. However, $Subgoal$ cannot be in $F_{I_S}^{\downarrow 1}$ since $Subgoal_{neg}$ is in $F_{I_S}^{\downarrow 0}$. Thus $Subgoal$ cannot be in F_N, nor can any subgoal in $U_{\mathcal{S}}$ since each of these subgoals depends on $Subgoal$. By the argument of the previous paragraph, the subgoals in $U_{\mathcal{S}}$ cannot be in T_{J_N} either, so they cannot be in J_N. This contradicts the assumption that the subgoal has dynamic stratum N; thus, the evaluation cannot in fact encounter a flummoxed state in a LRD-stratified program. $\diamondsuit$

A.2 Left-to-Right Weak Stratification

In order to show the power of LRD-stratification, we briefly present the notion of left-to-right weak stratification (LRW-stratification) based on [44], and show that the class of LRD-stratified programs properly includes that of LRW-stratified programs. We begin with some preliminary definitions.

Definition A.2.1 (*Atom Dependency Graph*) [44] *For a program P, the vertices of the* atom dependency graph G_P *are the ground atoms appearing (possibly negated) in P. The edges of G_P are directed and labeled either* positive *or* negative. *For every ground instantiation of a rule*

$$H \leftarrow B_1, \ldots, B_n.$$

in P, there are n edges in G_P. For $i = 1, \ldots, n$, if B_i is an atom then there is a positive edge from B_i to H in G_P; if B_i is a negated atom, say $\neg C_i$, then there is a negative edge from C_i to H in G_P.

Atom dependency relations $\leq_P$ *and* $<_P$ *between ground atoms of P are defined as:*

- *$B \leq_P A$ iff there is a directed path from B to A in G_P.*
- *$B <_P A$ iff there is a directed path from B to A in G_P passing through a negative edge.* $\square$

Note that in the above definition there can be both a positive and a negative edge between two ground atoms. The underlying idea behind weak stratification is to introduce the notion of the *components* of the atom dependency graph G_P and substitute them in for the *vertices* in the usual definitions of stratification or local stratification.

Definition A.2.2 (*Component*) [44] *Let $\sim_P$ be the equivalence relation between ground atoms defined as follows:*

$$A{\sim_P}B \text{ if and only if } (A = B) \vee (A{<_P}B \wedge B{<_P}A)$$

We shall refer to the equivalence classes introduced by $\sim_P$ as components *of G_P. A component of G_P is* trivial *iff it consists of only one element, say A, and $A{\not<_P}A$.* □

The relation $\sim_P$ denotes either equality of mutual negative dependence. In other words, according to the above definition, two distinct ground atoms A and B are equivalent if they are related by *mutual negative recursion*, i.e. recursion passing through negative literals.

Let C_1 and C_2 be two components of G_P. We define $C_1{\prec_P}C_2$ if and only if $C_1 \neq C_2$ and there exist ground atoms $A_1 \in C_1$ and $A_2 \in C_2$ such that $A_1{<_P}A_2$. A component C is called *minimal* if there is no component C' such that $C'{\prec_P}C$. For a program P, we define the *bottom stratum* $S(P)$ to be the union of all minimal components with respect to $\prec_P$. We define the *bottom layer* $L(P)$ of P to be the set of rules from P whose heads belong to $S(P)$. Herbrand models of $L(P)$ are identified with subsets $M \subseteq S(P)$ of the bottom stratum $S(P)$.

In order to define the class of left-to-right weakly stratified programs, we include the notion of a rule prefix from [56].

Definition A.2.3 (*Rule Prefix*) *A* rule prefix *is formed from a rule with n subgoals in the body by deleting the rightmost m subgoals, where $0 \leq m < n$.* □

Definition A.2.4 (*Reduction of* P *modulo* M) *Let $M \subseteq S(P)$ be a partial interpretation of a logic program P, that is, a consistent set of literals whose atoms are in $\mathcal{B}_P$. By the* reduction of P modulo M *we mean a new program $\frac{P}{M}$ obtained from P by performing the following operations:*

1. *Remove from P all rules that contain a* failing rule prefix*: for all j $(1 \leq j \leq i-1)$, L_j is true in M, and L_i is false in M.*

2. *Remove from P all rules whose head belongs to M.*

3. *Remove from all the remaining rules those literals that are members of M.*

4. *Finally, from the program resulting from the above operations, delete all rules with non-empty bodies whose heads appear in the program as unit facts.* □

The above definition of reduction differs from that given in [44] only in the addition of the failing rule prefix condition in step 1. Given the modified reduction operator, construction of the LRW-stratified model is identical to that of the weakly stratified model.

Definition A.2.5 (*Left-to-Right Weak Stratification*) [44] *Let P be a program; let $P_0 = P$; and let $M_0 = \emptyset$. Suppose that $\alpha > 0$ is a countable ordinal such that programs P_δ and partial interpretations M_δ have already been defined for all $\delta < \alpha$. Let*

$$N_\alpha = \bigcup_{0 \leq \delta < \alpha} M_\delta$$

and let $P_\alpha = \frac{P}{N_\alpha}$ be the reduction of P with respect to N_α.

- *If the program P_α is empty, then the construction stops and the program P is* left-to-right weakly stratified*. Then, N_α is the two valued model of P, and any ground atom that appears neither positively nor negatively in N_α is assumed false.*
- *Otherwise, if the bottom stratum $S(P_\alpha)$ of P_α is empty or if it contains a nontrivial component, then the construction stops. In such a case, P is not weakly stratified.*
- *Otherwise, the partial interpretation M_α is defined as the least model (restricted to the literals whose atoms are in $S(P_\alpha)$) of the bottom layer $L(P_\alpha)$ of P_α, and the construction continues.* □

Theorem A.2.1 (Theorem 3.2.3) *The class of left-to-right dynamically stratified programs properly includes the class of left-to-right weakly, and (left-to-right) modularly stratified programs.*

Proof: (Sketch) The proof that every LRW-stratified program is LRD-stratified, uses an induction on the number of reductions needed to construct the LRW-stratified model, and is a straightforward application of the definitions in this section and Section 3.2.1. The program of Example 3.1.2 then can be used to show that the inclusion is proper.

Proof of inclusion of left-to-right modularly stratified programs within LRW-stratified programs is a straightforward application of the definitions given here together with definitions in [56]. ♢

Appendix B

SLG-WAM: A Global View

B.1 Data Structures of the SLG-WAM

Since Chapters 4, 5, and 6 presented the development of the SLG-WAM in an incremental way, some of the main data structures of the abstract machine for SLG resolution were only presented in part. The following four figures (59, 58, 60, and 61) present the formats of generator choice points, subgoal, active subgoal, and negation suspension frames, which were also partly presented as Figures 8, 10, 11, and 28, respectively. We invite the reader to contrast them. All remaining SLG-WAM data structures remain as presented.

NextSF	Next Subgoal Frame
PreviousSF	Previous Subgoal Frame
Compl	Associated Completion Stack Frame
AnsTrieRoot	Root of the Answer Trie
AnsRetListH	Head of the Answer Return List
AnsRetListT	Tail of the Answer Return List
NIDEs	List of Negative IDEs of the Subgoal
ASFList	List of Active Subgoal Frames
NSFList	List of Negation Suspension Frames

Figure 58: *Format of Subgoal Frames in the SLG-WAM.*

We also note that the middle part of generator choice points is the same as the format of the "basic" SLG-WAM choice point; in particular, it is the format of Prolog choice point frames.

B.2 SLG-WAM instructions

Conditional answers require additional handling by the `new_answer` and `answer_return` instructions to that presented in Section 4.5. For example, as mentioned in Section 6.5.2, the

A_n	Argument Register n
$\vdots$	$\vdots$
A_1	Argument Register 1
FailCont	The Backward Continuation
EBreg	Environment Backtrack Point
Hreg	Top of Global Stack (Heap)
TRreg	Top of Trail
CPreg	Return point of call to predicate
Ereg	Parent Environment
PTreg	Parent Tabled Choice Point
Dreg	Delay List
PrevCP	Previous Choice Point
Subgoal	Pointer to the Subgoal Frame
BFreg	Choice Point Freeze Register
HFreg	Heap Freeze Register
TRFreg	Trail Freeze Register
EFreg	Local Stack Freeze Register

Figure 59: *Generator (Tabled) Choice Point in the SLG-WAM.*

FailCont	The Failure Continuation
EBreg	Environment Backtrack Point
Hreg	Top of Global Stack (Heap)
TRreg	Top of Trail
CPreg	Return point of call to predicate
Ereg	Parent Environment
PTreg	Parent Tabled Choice Point
Dreg	Delay List
PrevCP	Previous Choice Point
Subgoal	Pointer to the Subgoal Frame
LastAnswer	Last Answer that was Consumed
PrevASF	Previous Active Subgoal Frame

Figure 60: *Format of Active Subgoal Frames in the SLG-WAM.*

EBreg	Environment Backtrack Point
Hreg	Top of Global Stack (Heap)
TRreg	Top of (Forward) Trail Stack
CPreg	Return point of suspended literal
Ereg	Parent Environment
PTreg	Parent Tabled Choice Point
Dreg	Delay List
Subgoal	Frame of Suspended Subgoal
StratStatus	Stratification Status of the Suspension
PrevNSF	Previous Negation Suspension Frame

Figure 61: *Format of Negation Suspension Frames in the SLG-WAM.*

derivation of an unconditional answer for a subgoal S may cause the *early completion* of S and may trigger SIMPLIFICATION operations (cf. page 96).

The complete pseudocode for the new_answer instruction is presented in Figure 62. Similarly to the new_answer instruction for definite programs (cf. page 55), v_{n+1} is the extra permanent variable pointing to the generator choice point of the subgoal S for which a new answer A with *answer substitution* η_A and *delay list* DL_A has been derived and is about to be recorded in the appropriate answer table. The answer is inserted in the table only if it is still supported; i.e. none of the elements of DL_A is false in the current SLG system. This check is performed as part of interning DL_A. IDL_A points to the associated entry of the *IDL Table*, or indicates that the newly derived answer is unsupported and should be discarded.

<u>Instruction new_answer</u>($Arity$,v_{n+1})
    S := get_subgoal_from_pvar(v_{n+1});
    $answer\_table$:= SF_AnsTrieRoot(S);
    η_A := locate_substitution_factor($Arity$,v_{n+1});
    /* DL_A is the delay list pointed by the current value of **D** register */
    $A := \langle \eta_A; DL_A \rangle$;
    if ((IDL_A := intern(DL_A)) $\neq$ unsupported &&
        answer_check_insert(A, $answer\_table$) = new)   /* A was added */
1.   restore **D** register from the *Dreg* field of the generator CP of S;
2.   if ($IDL_A \neq [\,]$)   /* A is conditional */
2.1   add a positive delayed literal $\langle S, S\eta_A, \eta_A \rangle$ to the head of the delay list
        pointed by the current value of **D** register (the value just restored);
      else
2.2   initiate a simplify_pos_unconditional(η_A) SIMPLIFICATION;
2.3   if (A is a variant of S)   /* perform possible early completion */
          mark subgoal S as complete;
          initiate a simplify_neg_succeeds(S) SIMPLIFICATION;
3.   deallocate local environment;
4.   set the program pointer **P** to the continuation pointer **CP**;
    else
      fail;   /* A was either unsupported or already present in the table */

Figure 62: *The* new_answer *instruction.*

Recall that the new_answer instruction also performs *first-call optimization*, i.e. it returns the answer to the consuming node associated with the generator. If the new answer is conditional, the delay list of the generator node is extended by a new delay element: $\langle S, S\eta_A, \eta_A \rangle$ (step 2.1 in Figure 62). Finally, the answer_return instruction, which for definite programs was presented in Figure 17, is also modified to perform a similar operation when the answer that is being returned to an active node is conditional.

B.3 Remaining Simplification Procedures

Figure 63 presents pseudocode for a procedure that simplifies negative delay elements of a failing subgoal, while Figure 64 presents the procedure that deletes delay lists containing a positive delay element of an answer substitution whose status changed from supported to unsupported.

```
Procedure simplify_neg_fails(Subgoal)
 For each NIDE in the IDE list of Subgoal
   For each IDL in which this NIDE appears
     IDL := IDL − {NIDE}; /* remove the element from the delay list */
     if (IDL = []) then        /* answers are now unconditional */
       For each answer A having answer substitution ηS and IDL as delay list
         i. Initiate a simplify_pos_unconditional(ηS) simplification;
        ii. if (the subgoal S containing A now succeeds) then
              Initiate a simplify_neg_succeeds(S) simplification;
```

Figure 63: *Pseudocode for deletion of negative delay elements of a false subgoal.*

```
Procedure simplify_pos_unsupported(AnswerSubstitution)
 For each PIDE in the IDE list of the AnswerSubstitution
   For each IDL in which this PIDE appears
     Remove IDL from the IDL Table;        /* IDL contains a false literal */
     For each answer substitution ηS such that there is an answer A = ⟨ηS; IDL⟩
       Remove IDL from the set of delay lists of ηS;
       if (the resulting set is empty)            /* the status of ηS is */
          i. Delete ηS;                           /* now unsupported */
         ii. Initiate a simplify_pos_unsupported(ηS) simplification;
        iii. if (the subgoal S that had ηS as an answer substitution now fails)
               Initiate a simplify_neg_fails(S) simplification;
```

Figure 64: *Pseudocode for a simplification procedure that deletes delay lists.*

Bibliography

[1] K. Apt, H. Blair, and A. Walker. Towards a Theory of Declarative Knowledge. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 89–148. Morgan-Kaufmann, Los Altos, 1988.

[2] K. R. Apt and R. Bol. Logic Programming and Negation: A Survey. *Journal of Logic Programming*, 19 & 20:9–72, May 1994.

[3] C. Baral and V. S. Subrahmanian. Dualities between Alternative Semantics for Logic Programming and Non-monotonic Reasoning. *Journal of Automated Reasoning*, 10(3):399–420, June 1993.

[4] C. Beeri and R. Ramakrishnan. On the Power of Magic. *Journal of Logic Programming*, 10(3 & 4):255–299, April/May 1991.

[5] N. Bidoit. Negation in Rule-Based Database Languages: A Survey. *Theoretical Comput. Sci.*, 78(1):3–83, January 1991.

[6] N. Bidoit and C. Froidevaux. Negation by default and unstratifiable logic programs. *Theoretical Comput. Sci.*, 78(1):85–112, January 1991.

[7] N. Bidoit and P. Legay. WELL!: An Evaluation Procedure for All Logic Programs. In S. Abiteboul and P. C. Kanellakis, editors, *Proceedings of the Third International Conference on Database Theory*, number 470 in LNCS, pages 335–348, Paris, France, December 1990. Springer Verlag.

[8] 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.

[9] R. Bol and L. Degerstedt. The underlying search for magic templates and tabulation. In D. S. Warren, editor, *Proceedings of the Tenth International Conference on Logic Programming*, pages 793–811, Budapest, Hungary, June 1993. The MIT Press.

[10] W. Chen. Extending Prolog with Nonmonotonic Reasoning. *Journal of Logic Programming*, 27(2):169–183, May 1996.

[11] W. Chen, M. Kifer, and D. S. Warren. HiLog: A foundation for higher-order logic programming. *Journal of Logic Programming*, 15(3):187–230, February 1993.

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

[13] W. Chen and D. S. Warren. Computation of Stable Models and its Integration with Logical Query Evaluation. *IEEE Transactions on Knowledge and Data Engineering*, 1996. To appear. Also available as a technical report, Dept. of Computer Science, SUNY at Stony Brook.

[14] W. Chen and D. S. Warren. Tabled Evaluation with Delaying for General Logic Programs. *J. ACM*, 43(1):20–74, January 1996.

[15] P. Cholewiński, W. Marek, A. Mikitiuk, and M. Truszczyński. Experimenting with Nonmomotonic Reasoning. In L. Sterling, editor, *Proceedings of the 12th International Conference on Logic Programming*, pages 267–281, Japan, June 1995. The MIT Press.

[16] K. L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Presss, New York, 1978.

[17] D. Comer and R. Sethi. The complexity of trie index construction. *J. ACM*, 24(3):428–440, July 1977.

[18] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.

[19] S. Dawson, C. R. Ramakrishnan, and D. S. Warren. Practical Program Analysis Using General Purpose Logic Programming Systems — A Case Study. In *Proceedings of Conference on Programming Language Design and Implementation*, pages 117–126, Philadelphia, Pennsylvania, May 1996. ACM.

[20] L. Degerstedt and U. Nilsson. Magic Computation for Well-founded Semantics. In J. Dix, L. M. Pereira, and T. C. Przymusinski, editors, *Non-Monotonic Extensions of Logic Programming*, number 927 in LNAI, pages 181–204, Santa Margherita Ligure, Italy, June 1994. Springer-Verlag.

[21] S. Dietrich. *Extension Tables for Recursive Query Evaluation*. PhD thesis, Department of Computer Science, SUNY at Stony Brook, 1987.

[22] 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.

[23] J. Dix. Semantics of Logic Programs: Their Intuition and Formal Properties. An Overview. In A. Fuhrmann and H. Rott, editors, *Logic, Action, and Information. Proceedings of Konstanz Colloquium in Logic and Information (LogIn '92)*, pages 241–329. DeGruyter, 1993.

[24] W. F. Dowling and J. H. Galler. Linear-time algorithms for testing the satisfiability of propositional Horn formulae. *Journal of Logic Programming*, 1(3):267–284, 1984.

[25] F. Fages. A new Fixpoint Semantics for General Logic Programs compared with the Well-Founded and Stable Model Semantics. *New Generation Computing*, 9(3 & 4):425–443, 1991.

[26] M. Fitting. A Kripke-Kleene Semantics for Logic Programs. *Journal of Logic Programming*, 2(4):295–312, December 1985.

[27] J. Freire, T. Swift, and D. S. Warren. Beyond Depth-First: Improving Tabled Logic Programs through Alternative Scheduling Strategies. In H. Kuchen and D. Swierstra, editors, *Proceedings of the Eighth International Symposium on PLILP*, LNCS, Aachen, Germany, September 1996. Springer Verlag. To appear.

[28] M. Gelfond. On Stratified Autoepistemic Theories. In *Proceedings of AAAI-87*, pages 207–211, Seattle, Washington, July 1987. Morgan-Kaufmann.

[29] 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.

[30] A. C. Kakas and P. Mancarella. Preferred Extensions are Partial Stable Models. *Journal of Logic Programming*, 14(3 & 4):341–348, November 1992.

[31] D. B. Kemp and R. W. Topor. Completeness of a top-down query evaluation procedure for stratified databases. In R. A. Kowalski and K. A. Bowen, editors, *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pages 178–194, Seattle, August 1988. The MIT Press.

[32] M. Kifer, G. Lausen, and J. Wu. Logical Foundations of Object-Oriented and Frame-Based Languages. *J. ACM*, 42(4):741–843, July 1995.

[33] P. G. Kolaitis. The Expressive Power of Stratified Programs. *Information and Computation*, 90(1):50–66, January 1991.

[34] K. Kunen. Negation in Logic Programming. *Journal of Logic Programming*, 4(4):289–308, December 1987.

[35] R. Larson, D. S. Warren, J. Freire, and K. Sagonas. *Syntactica.* The MIT Press, Cambridge, MA, 1996.

[36] R. Larson, D. S. Warren, J. Freire, and K. Sagonas. *Semantica.* The MIT Press, Cambridge, MA, 1997. To appear.

[37] V. Lifschitz. On the Declarative Semantics of Logic Programs with Negation. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 177–192. Morgan-Kaufmann, Los Altos, 1988.

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

[39] W. Marek and M. Truszczyński. Autoepistemic Logic. *J. ACM*, 38(3):588–619, July 1991.

[40] S. Morishita. An Extension of Van Gelder's Alternating Fixpoint to Magic Programs. *J. Comput. Syst. Sci.*, 52(3):506–521, June 1996.

[41] J. F. Naughton, R. Ramakrishnan, Y. Sagiv, and J. D. Ullman. Argument Reduction by Factoring. *Theoretical Comput. Sci.*, 146(1 & 2):269–310, July 1995.

[42] I. Niemelä and P. Simons. Efficient Implementation of the Well-Founded and Stable Model Semantics. In M. Maher, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, Bonn, Germany, September 1996. The MIT Press. To appear.

[43] L. M. Pereira, J. N. Aparício, and J. J. Alferes. Non-monotonic Reasoning with Logic Programming. *Journal of Logic Programming*, 17(2, 3, & 4):227–263, November 1993.

[44] H. Przymusinska and T. C. Przymusinski. Weakly Stratified Logic Programs. *Fundamenta Informaticae*, 13(1):51–65, March 1990.

[45] 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.

[46] T. C. Przymusinski. On the Declarative and Procedural Semantics of Logic Programs. *Journal of Automated Reasoning*, 5(2):167–205, June 1989.

[47] T. C. Przymusinski. The Well-founded Semantics Coincides with the Three-valued Stable Semantics. *Fundamenta Informaticae*, 13(4):445–463, December 1990.

[48] T. C. Przymusinski. Three-valued Non-Monotonic Formalisms and Semantics of Logic Programs. *Journal of Artificial Intelligence*, 49(1–3):309–343, January 1991.

[49] T. C. Przymusinski. Well-Founded and Stationary Models of Logic Programs. *Annals of Mathematics and Artificial Intelligence*, 12:141–187, 1994.

[50] I. V. Ramakrishnan, P. Rao, K. Sagonas, T. Swift, and D. S. Warren. Efficient Tabling Mechanisms for Logic Programs. In L. Sterling, editor, *Proceedings of the 12th International Conference on Logic Programming*, pages 687–711, Japan, June 1995. The MIT Press.

[51] R. Ramakrishnan. Magic Templates: A Spellbinding Approach To Logic Programs. *Journal of Logic Programming*, 11(3 & 4):189–216, October/November 1991.

[52] 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.

[53] 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.

[54] K. A. Ross. *The Semantics of Deductive Databases.* PhD thesis, Department of Computer Science, Stanford University, August 1991.

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

[56] K. A. Ross. Modular Stratification and Magic Sets for Datalog programs with Negation. *J. ACM*, 41(6):1216–1266, November 1994.

[57] D. Saccà and C. Zaniolo. Stable Models and Non-Determinism in Logic Programs with Negation. In *Proceedings of the Ninth ACM on Principles of Database Systems*, pages 205–217, Nashville, Tennessee, April 1990. ACM.

[58] 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.

[59] K. Sagonas, T. Swift, and D. S. Warren. An Abstract Machine for Computing the Well-Founded Semantics. In M. Maher, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, Bonn, Germany, September 1996. The MIT Press. To appear.

[60] K. Sagonas, T. Swift, and D. S. Warren. An Abstract Machine for Fixed-Order Dynamically Stratified Programs. In M. A. McRobbie and J. K. Slaney, editors, *Proceedings of the Thirteenth International Conference on Automated Deduction*, number 1104 in LNAI, pages 328–342, New Brunswick, New Jersey, July/August 1996. Springer-Verlag.

[61] K. Sagonas, T. Swift, and D. S. Warren. An Abstract Machine for Fixed-Order Stratified Logic Programs. Submitted for publication, 1996.

[62] K. Sagonas, T. Swift, and D. S. Warren. The Limits of Fixed-Order Computation. In D. Pedreschi and C. Zaniolo, editors, *Proceedings of the First International Workshop on Logic in Databases*, LNCS, San Miniato, Italy, July 1996. Springer-Verlag. To appear.

[63] H. Seki. On the Power of Alexander Templates. In *Proceedings of the Eighth ACM Symposium on Principles of Database Systems*, pages 150–159, Philadelphia, Pennsylvania, March 1989. ACM.

[64] H. Seki and H. Itoh. A query evaluation method for stratified programs under the extended CWA. In R. A. Kowalski and K. A. Bowen, editors, *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pages 195–211, Seattle, August 1988. The MIT Press.

[65] J. C. Shepherdson. Negation in Logic Programming for General Logic Programs. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 19–88. Morgan-Kaufmann, Los Altos, 1988.

[66] 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.

[67] S. Sudarshan and R. Ramakrishnan. Optimizations of Bottom-Up Evaluation with Non-Ground Terms. In D. Miller, editor, *Proceedings of the 1993 International Symposium on Logic Programming*, pages 557–574, Vancouver, October 1993. The MIT Press.

[68] T. Swift. *Efficient Evaluation of Normal Logic Programs*. PhD thesis, Department of Computer Science, SUNY at Stony Brook, December 1994.

[69] 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.

[70] T. Swift and D. S. Warren. Analysis of SLG-WAM Evaluation of Definite Programs. In M. Bruynooghe, editor, *Proceedings of the 1994 International Symposium on Logic Programming*, pages 219–235, Ithaca, November 1994. The MIT Press.

[71] H. Tamaki and T. Sato. OLD Resolution with Tabulation. In E. Shapiro, editor, *Proceedings of the Third International Conference on Logic Programming*, number 225 in LNCS, pages 84–98, London, July 1986. Springer-Verlag.

[72] M. H. van Emden and R. A. Kowalski. The semantics of predicate logic as a programming language. *J. ACM*, 23(4):733–742, October 1976.

[73] A. Van Gelder. Negation as Failure using Tight Derivations for General Logic Programs. *Journal of Logic Programming*, 6(1 & 2):109–134, January/March 1989.

[74] 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.

[75] 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.

[76] P. Van Roy. 1983–1993: The wonder years of sequential Prolog implementation. *Journal of Logic Programming*, 19/20:385–441, May/July 1994.

[77] L. Vielle. Recursive Query Processing: The Power of Logic. *Theoretical Comput. Sci.*, 69(1):1–53, December 1989.

[78] E. Villemonte de la Clergerie. Layer Sharing: an improved Structure-Sharing Framework. In *Proceedings of the ACM Conference on Principles of Programming Languages*, pages 345–359, Charleston, South Carolina, January 1993.

[79] A. Walker. Backchain Iteration: Towards a practical inference method that is simple enough to be proved terminating, sound, and complete. *Journal of Automated Reasoning*, 11(1):1–23, August 1993.

[80] D. H. D. Warren. An abstract Prolog instruction set. Technical Report 309, SRI International, Menlo Park, U.S.A., October 1983.

[81] D. S. Warren. Efficient Prolog memory management for flexible control strategies. In *Proceedings of the 1984 International Symposium on Logic Programming*, pages 198–202, Atlantic City, February 1984.

[82] S. Zhang, S. A. Smolka, and O. Sokolsky. On the parallel complexity of model checking in the modal mu-calculus. In *Proceedings of the IEEE Symposium on Logic in Computer Science*, pages 154–163, Paris, France, July 1994. IEEE Computer Science Press.