THE SEMANTICS OF DEDUCTIVE DATABASES

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

By
Kenneth Andrew Ross
August 1991

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Jeffrey D. Ullman
(Principal Adviser)

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

John Mitchell

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Phokion Kolaitis
(U. C. Santa Cruz)

Approved for the University Committee on Graduate Studies:

Dean of Graduate Studies

Abstract

The major advantage of a deductive database is the ability to write queries and programs declaratively, using both facts and simple logical rules to represent knowledge. Declarativeness makes queries easier to write, and thus reduces the time taken and the programming skills needed to specify a query. Two important questions arise: What should the semantics of such a deductive database language be, and how should declarative queries be answered efficiently? We address both of these questions.

The question of semantics becomes complicated when one needs some sort of negation by default. For example, the absence of a fact stating that a student is in a class should allow us to conclude that the student is not in that class. Several researchers have previously looked at this problem and have tried to come up with formal definitions of an intuitively satisfying semantics. Unfortunately, they were either too weak and didn't allow sufficiently many conclusions to be drawn, or they were only well-defined for a restricted class of programs. Our first contribution is a semantics that we call the "well-founded semantics" in which all programs can be accomodated in a satisfying manner, even when the rules are recursive through negation. This semantics uses a three-valued logic in which propositions may be true, false or undefined. We study various properties of this semantics, and show how it generalizes previous approaches. We also investigate how it extends to programs with additional features, such as second order constructs.

As part of the second question, we look at various ways of evaluating the well-founded semantics. We propose a top-down method that is sound and complete with respect to the well-founded semantics. We investigate optimization techniques such as magic sets and show how to apply them to a large class of programs that we call "modularly stratified." Our magic sets method has been implemented as part of the NAIL! system at Stanford. We also investigate other optimization techniques, such as tail-recursion elimination.

Preface

This thesis is based on five papers written while the author was at Stanford University [94, 96, 97, 98, 118]. Chapter 3 is based on [118], a joint paper written together with Allen Van Gelder and John Schlipf. Some material in Chapter 3 is taken directly from that paper, with the permission of my coauthors. Those results in [118] that were not at least partly due to me have been either omitted or summarized with attribution. The other papers were single-author papers.

Acknowledgements

I would firstly like to thank my advisor, Jeff Ullman, for his guidance during my four years at Stanford. He allowed me to work in a relatively independent fashion, while always providing valuable advice and suggestions. I would also like to thank my readers, Phokion Kolaitis and John Mitchell, for their enthusiasm and their helpful suggestions.

I am very grateful to the other members of the NAIL! group at Stanford for providing a stimulating environment in which to work. In particular, I would like to thank Surajit Chaudhuri, Marcia Derr, Ashish Gupta, Håkan Jakobsson, Kate Morris, Inderpal Singh Mumick, Geoff Phipps, Shuky Sagiv, Yatin Saraiya, Alberto Torres, and Moshe Vardi.

I would like to thank Allen Van Gelder, who recognized that my undergraduate thesis work [100] could be relevant for defining a better semantics for deductive databases and logic programs. He and John Schlipf coauthored with me the paper on the well-founded semantics that opened the way for this thesis.

In addition to those people mentioned above, various people have provided valuable comments on the work presented in this thesis. For such comments I would like to thank Michael Kifer, Vladimir Lifchitz, Michael Maher, Teodor Przymusinski, Rodney Topor and a number of anonymous referees.

I would like to thank my family for their support and encouragement for me to pursue a PhD in the United States. And last, but certainly not least, I would like to thank my wife Sylvia for her loving support.

During my time at Stanford, I have been supported by NSF grant IRI-87-22886, a grant from IBM corporation, by the AFOSR under contract number 88-0266, and by an IBM Fellowship. Part of the work in Chapters 6 and 7 was performed while visiting the IBM T. J. Watson Research Laboratory.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Deductive Databases and Declarativeness

Declarativeness is often cited as a desirable feature of a language, be it a programming language, a database language, a language for reasoning about programs, a language for reasoning about knowledge, or a language for expressing information about verification. Roughly, declarativeness means that the person writing in that language specifies "what to compute" and not "how to compute it."

For discussion of the relative merits of declarative and procedural (or imperative) systems, see [1, 39, 40, 55, 69, 113].

In principle, declarativeness requires less of a user or programmer, and can thus be the basis of a system in which writing queries can be significantly less time-consuming than with a procedural language. Empirical results along these lines (for older database technology) have been reported in [111].[1]

Of course, declarativeness is not a black-or-white issue. Different people have different tastes in this respect. Compared with previous approaches, the database language SQL was far more declarative, and became widely accepted at least partly because of this feature. However, many modern database researchers would debate the declarativeness of SQL, especially as it is widely used today embedded inside a procedural programming language.

Because SQL has limited expressive power, (for example it cannot express transitive closures), embedding SQL in a fully fledged programming language has been necessary in practice. There are two problems with using a language like SQL within a procedural language. One is the loss of overall declarativeness. The second is known as the "impedance mismatch" problem.

The impedance mismatch problem occurs when a database system is embedded inside a language that is not primarily a database language. The data types used in the database system and those used in the outer programming language are likely to be very different, causing confusion for programmers and substantial system overhead for data conversion. Even more serious, though, is that potential for optimization is lost. An autonomous database may batch

[1] It is also shown in [111] that coding queries in the procedural language produced queries that could be evaluated faster than the corresponding queries in the declarative language. It is unclear whether this comparison reflects well on the programmers in the study, or badly on the optimizer used in their declarative system. In any case, the time to answer queries was on the order of seconds for each query studied, while coding time for the declarative system could be measured in minutes, and coding time for the procedural system was measured in hours.

subqueries in such a way that repeated computation is minimized. Making multiple calls to a database system from a programming language, with each call returning a small fragment of the information needed to answer the query, is a strategy likely to lead to significant repeated computation between calls, not to mention the overhead of starting up the database on each call.

The approach of the NAIL! system at Stanford has been to use a declarative language (NAIL) together with a procedural language (Glue). NAIL itself is significantly more powerful that SQL, and much more can be done within the declarative language itself. Glue was designed with the goal of minimizing the impedance mismatch problem while providing the nondeclarative features necessary in a complete system. One would write queries in NAIL, resorting to Glue for updates, I/O, and when specialized algorithms need to be implemented. We shall not discuss the Glue language in this thesis; the interested reader is referred to [80]. In this thesis we shall concentrate on an abstract declarative component for a database system.

The model for declarative specification that we choose is that of a deductive database. Our queries and our database will be expressed in terms of logical rules. Not surprisingly, our semantic results will be phrased in the language of first order predicate logic (with some extensions to a higher order logic in Chapter 8). Our choice of the logical model relies on its mathematical clarity, and thus its apparent declarativeness. This is not the only choice for a declarative system. For example, many researchers are currently studying declarative ways of specifying object-oriented systems.

Deductive databases contain *facts* and *rules*. Facts are simply tuples; present-day relational systems may be thought of as sets of facts. Rules can define views on the database. For example, suppose that the relation *marriage* has four attributes: male partner, female partner, date of marriage, location of marriage. So we may have the following *marriage* facts.

$$\begin{array}{l} \mathit{marriage}(\mathit{john}, \mathit{mary}, \text{June 1 1950}, \mathit{San\ Francisco}) \\ \mathit{marriage}(\mathit{fred}, \mathit{joan}, \text{July 7 1978}, \mathit{San\ Jose}) \end{array}$$

Suppose we want to know those people who are married. One approach might be to write a program that scans through the *marriage* relation accumulating those names appearing in the first or second arguments. The disadvantage of this approach is that there is no transparent way to make use of the information in computing more complicated queries involving married people. In a deductive database, we may define the *married* relation by the following rules.

$$\begin{array}{l} \mathit{married}(X) \leftarrow \mathit{marriage}(X, Y, Z, W) \\ \mathit{married}(Y) \leftarrow \mathit{marriage}(X, Y, Z, W) \end{array}$$

We may then use subgoals involving the predicate *married* on the right hand side of other rules. The *married* relation above may be thought of as a view on the database. If the *marriage* relation were to change, say because of a divorce or death, then the view provided by the *married* relation would be consistent with the new information, without any explicit change to the *married* relation. This particular view could be expressed in relational calculus as

$$\{X : \exists Y, Z, W(\mathit{marriage}(X, Y, Z, W) \vee \mathit{marriage}(Y, X, Z, W))\}$$

or in relational algebra as

$$\pi_1(\mathit{marriage}) \cup \pi_2(\mathit{marriage}).$$

Unfortunately, there are views we would like to express that cannot be expressed in relational algebra or relational calculus. One such example is transitive closure. Consider a directed graph represented by the binary relation e; $e(X,Y)$ holds if there is an edge from X to Y. The *transitive closure* of e is the path relation induced by e. We can represent the transitive closure using two rules one of which is recursive.

$$\begin{array}{l} tc(X,Y) \leftarrow e(X,Y) \\ tc(X,Z) \leftarrow e(X,Y), tc(Y,Z) \end{array}$$

It is precisely this recursiveness that makes the rules we use more expressive than relational algebra or relation calculus expressions [4]. (We shall be more precise about the nature of rules in Chapter 2.)

We shall also want to incorporate negation by default in our rules. For example we may use the rule

$$single(X) \leftarrow person(X), \neg married(X)$$

to determine people who are single. The important thing to note is our assumption that unless we can establish that $married(X)$ is true for some person X, we shall assume that $married(X)$ is false. Thus our deduction is not classical, since for classical logic we would explicitly have to include facts of the form $\neg married(X)$ for each person X that does not appear in the first or second argument of the *marriage* relation in order to fire the rule above. Including such negative information explicitly is undesirable because the amount of negative information is likely to be large, perhaps even an order of magnitude larger than the positive information in the database.[2]

When recursion is combined with negation by default, many semantic questions arise. For example, what is the meaning of the rule

$$p \leftarrow \neg p \quad ?$$

Providing answers to such questions is one of the main aims of this thesis.

1.2 What is a semantics?

In the context of programming languages, the aim of a semantics is to provide an account of what is a "correct" computation, independent of the computation mechanism itself. The motivation for a declarative semantics in the context of databases is similar. If the correctness of an answer is defined by a particular model, say, then we are free to choose any computation mechanism and assess how close it comes to computing the correct answer. This contrasts with operational semantics that resort to stating "Correct answers are those returned by the procedure." A desirable feature of a semantics is that it can be defined both operationally and declaratively. In other words, the semantics has a non-procedural interpretation as well as a computational mechanism that computes it.

In programming languages the question of semantics is complicated by several issues that we ignore for deductive databases. The first of these issues is the notion of state. The behavior of a programming language (with some exceptions) is usually described according to how it performs statement by statement, with state information being kept between statements. In this thesis

[2]For example, the set of students at a university *not* taking history-101 is likely to be much larger than the set of students taking the course.

we assume that the database is fixed, and that the programs are used simply to answer queries on views of the database rather than to change the database itself. Updates to the database do not concern us here, although they would certainly be an essential element of a database system, and may be implemented in a "lower-level" procedural language.[3]

Another such issue is side-effects. While programming languages typically allow changes to variables, input and output, our declarative language does not permit such operations in order to retain a declarative system in which subgoals can be reordered within rules, and rules themselves can be reordered without any change in semantics.[4] Again, in a complete system, these functions may be relegated to a lower-level language.

While we may sometimes use the word "logic program" or "program" as a substitute for "deductive database" we remind the reader that our focus will be within the context of deductive databases, not programming languages.

There have been a number of adjectives attached to the word "semantics" in deductive databases and logic programming. We shall briefly describe their nature below.

Model-theoretic A semantics is model-theoretic if it is defined in terms of one or several distinguished models of the program.

Proof-theoretic A semantics is proof-theoretic if it is defined as the consequences of some theory based on the program according to some set of axioms and inference rules.

Procedural A procedural semantics is one based on a procedure, stating effectively that correct answers are those returned by the procedure.

Declarative Typically, a declarative semantics is nonprocedural and formally defined in a mathematically precise manner.[5]

Fixpoint A fixpoint semantics is one defined by taking one or many fixpoints of a certain operator to be the distinguished models defining the semantics.

2-valued A 2-valued semantics is one that assigns to all sentences one of two truth values, *true* or *false*.

3-valued A 3-valued semantics is one that assigns to all sentences one of three truth values, *true*, *false*, or *undefined*.

Herbrand A semantics in which the domain is exactly those terms constructible from the constants and functions appearing in the program, with terms interpreted freely.

1.3 One Intuitive Semantics?

In this section, we will argue that there is no one "intuitive" or "natural" declarative semantics for logic-based programs. Different expectations of program behavior can lead to different interpretations of the same program. Within specific contexts, a number of authors have claimed

[3]This approach is taken by the NAIL! system at Stanford.

[4]Such reordering is frequently needed to optimize queries.

[5]Declarativeness is often a matter of taste, and there have been frequent debates about whether particular semantics are declarative or not. The basic requirement is that the semantics be understood independent of any computation mechanism.

that there is a single "natural" meaning for a program, and that their proposals are faithful to that meaning. All too often, authors are unaware of other contexts in which a program may be used, and fail to either justify their choice of "natural" semantics or identify the context in which it applies.

Often, examples accompanying a proposal use meaningless single-letter propositions that fail to describe the context under consideration. For example, is

$$a \leftarrow \neg b$$

equivalent to

$$b \leftarrow \neg a$$

and to

$$a \vee b \text{ ?}$$

It depends what our notion of equivalence is. We need a context for a and b. If we are doing classical theorem-proving, and a and b are propositions, then presumably we will expect the above equivalences to hold. If we are performing some sort of default reasoning where a is inferred if we fail to prove b, then we have an implicit direction to the implication and so we would not expect the above equivalences to hold.

We cannot expect the system to abstract context information from the predicate names (although the users may be able to), and so the context should be specified either explicitly to a system that then chooses an appropriate semantics, or implicitly in the choice of one appropriate semantics. This context information, rather than the form of the program itself, should guide the choice of an appropriate semantics.

Authors have been critical of alternative proposals that are not really competitors, since they are intended for use in a different context. In a sense it is a coincidence that similar syntax is chosen to model problems with different semantic intuitions. The sharing of syntax causes unnecessary confusion. For example, consider the simple program

$$p \leftarrow p.$$

A logic-programmer may reason as follows:

> This program is clearly an infinite loop. Calling p will result in another recursive call to p, continuing on indefinitely.

A person more interested in databases may say:

> This rule is a tautology that doesn't tell us anything. If we knew p then we would fire the rule and re-derive p, with duplicate elimination terminating the computation. If we did not know p then the rule doesn't fire, and that's as far as we get. We don't derive p and so p is clearly false.

The root of this problem is the different expectation that each person has of their semantics. The logic-programmer is modelling a top-down resolution-based method (such as Prolog), while the database programmer is modelling a bottom-up tuple-driven evaluation mechanism.

An interesting development was the modification of top-down methods to perform memoing for function-free programs. Memoing enables the detection of subgoals that have already been considered, and allows infinite computations (such as that generated by the $p \leftarrow p$ program above) to be pruned. A consequence of this pruning is that the query ?-p fails rather than recursing infinitely, and so a corresponding switch in semantics is necessary. The development of memoing was aimed at using techniques from logic programming in database applications; note how the semantics depends more upon the application than upon the nature of the procedure involved in the computation.

A result of the different expectations of declarative semantics for logic programs is a large number of papers on topics such as negation. On one hand it is desirable to have different semantics for different applications. We consider various proposals throughout this thesis, and would like to have some way of assessing how "successful" a semantics is. We discuss desirable features of a semantics in Section 1.4. It is clear from the preceding discussion that simply stating that a semantics is "intuitive" is not sufficient justification for that semantics. We may, however, take the liberty of labelling a semantics "intuitive" or "natural" within a particular context by demonstrating agreement with other, well-accepted proposals within that context, or by demonstrating that a semantics appropriately handles examples in the particular context.

On the other hand, for reasons of coherence, it is usually not desirable for a particular application to let programmers choose their own preferred semantics.

We now continue with some further examples of cases where different semantics give different results, but results that are still appropriate for the corresponding application. We consider the following types of logic-based application, each of which has its own requirements:

- Theorem Proving.
- Logic Programming.
- (Deductive) Databases.
- Production Systems.
- Common-sense (nonmonotonic) Reasoning.

Theorem Proving

Theorem proving is in a sense a special case. We do not usually want to infer *anything* false by default, since the absence of a proof for a proposition does not mean that the proposition is false. The semantics of what is *true* should match the logic in which the theorem-prover is working. For example, in intuitionistic logic we should not be able to derive p from the program

$$\begin{array}{l} p \leftarrow q \\ p \leftarrow \neg q \end{array}$$

even though p is a classical consequence of the program. Within a classical framework, however, we should derive p.

Logic Programming

Logic programming is a paradigm for general purpose programming. It's semantics, therefore, should correspond closely to the behavior of a logic-based programming language, such as a "pure" subset of Prolog. Thus, as discussed above, it is appropriate to make p undefined for the program

$$p \leftarrow p$$

since calling p yields an infinite loop. The program completion yields such a semantics for Horn programs. For normal programs, extensions of the completion approach by Fitting and Kunen provide a coherent three-valued semantics that matches the expectations of logic programmers.

Deductive Databases

Deductive databases present a different approach. Predicates may be thought of as views on relations. These views may be defined recursively, as in the transitive closure program

$$\begin{aligned} tc(X,Y) &\leftarrow e(X,Y) \\ tc(X,Z) &\leftarrow e(X,Y), tc(Y,Z) \\ u(X) &\leftarrow tc(X,Y), \neg tc(Y,X) \end{aligned}$$

Assume that there are also some tuples for the e relation. Think of the e relation as an edge relation of a directed graph. Then tc computes the path relation on that graph. The relation u contains every node a from which it is possible to reach another node b from which a is not reachable. This interpretation of u is the one defined by the "well-founded semantics" that is proposed in this thesis. Note that the completion-based approaches (briefly discussed above, and defined in Chapter 3) do not agree with the well-founded semantics here, in particular when the e relation has cycles.

The program above is "stratified," and several previous proposals would have given the same result as the well-founded semantics. However, unlike the well-founded semantics, the proposals based on stratification are not well defined for all normal programs. An example of a non-stratified program for which the previous proposals would not be well-defined is the game example

$$winning(X) \leftarrow move(X,Y), \neg winning(Y)$$

together with some tuples for the *move* relation. This program models a game in which the object is to force the opponent into a position from which there are no legal moves. Given a "game tree" represented in the *move* relation, the well-founded semantics makes $winning(a)$ true if a is a winning position; it makes $winning(a)$ false if a is a losing position; and it makes $winning(a)$ undefined if a is a drawn position. Note that for the game program above, $winning(a) \vee \neg winning(a)$ is not necessarily true, and so inferring p from the program above together with the rules

$$\begin{aligned} p &\leftarrow winning(a) \\ p &\leftarrow \neg winning(a) \end{aligned}$$

would not be correct if a was a drawn position. Thus the well-founded semantics has a constructive flavor; it does not necessarily infer all classical consequences of the program.

Production Systems

Like logic programming, production systems are used for general purpose programming. The semantics of production systems is typically given using an *inflationary* construction: A negative subgoal $\neg p$ succeeds if p is not *currently* true. If p is later derived then nothing is retracted. One form of inflationary semantics is given by firing all rules in parallel, one step at a time. Consider the program

$$\begin{aligned} tc(X,Y) &\leftarrow e(X,Y) \\ tc(X,Z) &\leftarrow e(X,Y), tc(Y,Z) \\ u(X) &\leftarrow tc(X,Y), \neg tc(Y,X) \end{aligned}$$

introduced above, together with some tuples for the e relation. The tc relation at the $n+1$st iteration contains those paths in the graph of length up to n. The third rule fires at the $n+1$st iteration generating $u(a)$, say, if there is a path from a to some node b of length at most n, but no such path from b to a. Thus, by the end of the computation, the relation u contains every node from which it is possible to reach another node in fewer hops than it would take from that other node to reach it. Contrast this interpretation with the database interpretation above that makes u contain every node from which it is possible to reach another node from which the original node is not reachable. The inflationary semantics is very sensitive to timing; substituting the pair $p \leftarrow q$, $q \leftarrow r$ for $p \leftarrow r$, where q is a new symbol, may change the semantics of the program entirely. This timing issue is a drawback that can make production systems difficult to deal with, and certainly less declarative than some of the other semantics described above. Nevertheless, production systems such as OPS5 are in common use. Also, on a theoretical level, the inflationary semantics has equivalent expressive power to the well-founded semantics.

Nonmonotonic Reasoning

Nonmonotonic reasoning is the subarea of artificial intelligence dealing with commonsense knowledge involving reasoning by default. Traditionally, the formalisms employed in nonmonotonic reasoning are significantly more general than the class of programs we consider. While more can be expressed, there is the tradeoff that computing the semantics becomes harder. The goals of nonmonotonic reasoning are not that far from those of deductive databases and logic programming. In fact for the class of programs we consider, there is a close link between the well-founded semantics and several formalisms in nonmonotonic reasoning. However, the range of phenomena that general commonsense reasoning aims to model usually enables one to come up with examples that are not correctly handled by a particular semantics.

The main focus of this thesis is deductive databases. Nevertheless, some of the results presented here are relevant to the areas of logic programming and nonmonotonic reasoning. In Chapter 3 where we introduce the well-founded semantics, we shall compare it with a variety of other proposals. In Chapter 4 we shall present a procedural semantics that is equivalent to the well-founded semantics. Chapters 5 and 6 discuss certain optimizations for query evaluation. In Chapter 7 we shall investigate certain semantic properties of a class of programs that we call "modularly acyclic." In Chapter 8 we look at issues of semantics and optimization in a class of programs having second order syntax.

1.4 What is a Good Semantics?

Our criteria for evaluating the utility of a semantics are listed below. A semantics should be

Decidable The semantics is computable.

Low Complexity The inference process is relatively efficient.

Comprehensible The semantics is easily understood.

Appropriate The semantics matches users' expectations on examples in the particular application area.

Applicable The semantics should be well-defined for as large a class of programs as possible.

Expressive Users should be able to (easily) express all queries that they need.

Compositional If P_1 is a program and P_2 is a program that uses (but does not define) some predicates from P_1, then the semantics for P_1 should extend to a semantics for $P_1 \cup P_2$.

We shall try to evaluate a semantics according to the criteria above, for appropriate classes of programs. Unfortunately, it is difficult to assess the criterion that a semantics be comprehensible.

Firstly, any semantics is comprehensible to those who propose it, and so it is up to potential users of a semantics to assess its comprehensibility. Secondly, even if we had a significant group of users for each semantics, the question of comprehensibility is a psychological one, and not one that the author is qualified to judge. While this issue is of some importance, and may provide interesting research for one involved in human factors and user interfaces, it is beyond the scope of this thesis.

There are two possible ways of interpreting the question of expressiveness. The theoretical approach is to construct methods by which one semantics can simulate a second, thus showing the first semantics to be at least as expressive as the second. The practical approach is to assess how "easy" it is to implement a given query according to two different semantics, and prefer the semantics that makes it easier to write most queries. There is often a discrepancy between the two approaches, since the simulation constructed by the theoretical approach can be intricate and obviously impractical. Conversely, there are no known rigorous ways of measuring how "easy" it is to implement a query in various environments. Further, since deductive databases are a relatively new technology, there are no empirical studies of such issues. We leave the task of performing such studies for future research.

There has been a reasonable amount of work done studying the relative (theoretical) expressiveness of different semantics, and we shall survey some of these results in this thesis. One should be careful, though, not to take these expressiveness results as practical guidelines for choosing a real-world semantics. For example, the class of function-free Horn-clause programs expresses a subset of the polynomial time queries. This class of programs *cannot* express the query "Does relation r contain an even number of tuples?" In practice, the solution would be to augment the language with features that enable one to count the number of tuples in a relation. Theoretically, though, this choice is unattractive because it extends the domain from being finite to being infinite (since it must contain the integers), and the integers can then be used to simulate Turing machines. The theoretical solution is thus different; an ordering of the domain elements is assumed to be given by some predefined predicate, and the query about even-ness can be answered by a program that uses this predicate.

In a reasonable large system, modularity is an important requirement. Modularity allows different programmers to define different parts of a database, with only the interfaces between parts having to be known to others. In the context of deductive databases, one prerequisite for modularity is compositionality. Suppose that I write a program to take the transitive closure of an arbitrary edge relation e. My program should work irrespective of the particular tuples in e, and irrespective of how the e relation was defined. Thus, one would like the composition[6] of an arbitrary program component defining the e relation with my transitive closure program to have the expected semantics. The criterion of compositionality is closely related to what Schlipf calls the "principle of stratification" in [104].

The are also tradeoffs among the other criteria. For example, the larger the class of programs being considered, or the more expressive the semantics, the higher the complexity of answering queries is likely to be.

[6] Think of composition as union, except that local predicates in each component are renamed to avoid name conflicts.

1.5 Optimization

Declarative systems such as deductive databases try to make posing queries simple by letting the users specify *what* they want to compute rather than *how* to compute it. This strategy leads to databases in which it is relatively easy to pose queries. However, since the user does not supply information on how to answer the query, the burden of deciding how to evaluate the query falls on the system.

Thus declarative systems must have powerful optimization techniques available. In large databases they could make the difference between a query being feasible or infeasible. For non-recursive queries many well-known optimizations for relational databases can be applied. Several techniques for optimizing recursive queries have been recently proposed, including "magic-sets," "right-linear evaluation," "counting methods" and so on. (See [113] for a comprehensive discussion of these and other strategies.)

Magic-sets [11, 13, 88, 93] is a general technique that can, in principle, be applied to any (recursive or nonrecursive) deductive database. By passing binding information from the query itself into the rule evaluation, magic sets restricts the computation to tuples that are in some sense relevant. In Chapter 5 we extend these magic-sets techniques to programs with (not necessarily stratified) negation.

Other techniques, such as right-linear evaluation [76, 77] give better performance than magic sets for restricted classes of programs. Common programs such as transitive closures fall into the class of right-linear programs. The right-linear optimization can be thought of as a (limited) form of tail-recursion elimination. In Chapter 6 we propose a method for performing tail-recursion elimination that is significantly more general than that for right-linear programs.

Chapter 2

Preliminaries

In this chapter we present some of the basic terminology used throughout this thesis. We assume some familiarity with first order logic, relational algebra, and relational calculus.

Most of the terminology used here is borrowed from Lloyd [60] and Ullman [113].

2.1 Terminology

In this section we introduce our notation and basic definitions, and describe the class of normal programs that we shall be considering in this thesis.

Definition 2.1.1: A *term* is defined inductively as follows:

- A variable or a constant symbol is a term.
- If f is an n-ary function symbol and $t_1, \cdots, t_n$ are terms then $f(t_1, \cdots, t_n)$ is a term.

If p is an n-ary predicate symbol and $t_1, \cdots, t_n$ are terms then $p(t_1, \cdots, t_n)$ is an *atom.* A *literal* is either an atom or a negated atom. When we write an atom $p(\vec{X})$ it is understood that $\vec{X}$ is a vector of terms, not necessarily variables. □

Definition 2.1.2: A *normal rule* is a sentence of the form

$$A \leftarrow L_1, \ldots, L_n$$

where A is an atom, and $L_1, \ldots, L_n$ are (positive or negative) literals. We refer to A as the *head* of the rule and $L_1, \ldots, L_n$ as the *body* of the rule. Each L_i is a *subgoal* of the rule. All variables are assumed to be universally quantified at the front of the clause, and the commas in the body denote conjunction. If the body of a rule is empty then we may refer to the rule as a *fact*, and omit the "$\leftarrow$" symbol.

A *normal program* is a finite set of normal rules. A *Horn rule* is one with no negative subgoals, and a *Horn program* is one with only Horn rules. A *Datalog* program is a function-free Horn program. □

Logical variables begin with a capital letter; constants, functions, and predicates begin with a lowercase letter. We use the same symbol, e.g., p, to refer to both a predicate and its relation. The word *ground* is used as a synonym for "variable-free."

Definition 2.1.3: The *Herbrand universe* of a program P is the set of all variable-free terms that may be formed from the constants and function symbols appearing in P. If there are no constants in P then we (arbitrarily) add a single extra constant symbol.

The *Herbrand base* of P is the set of atomic formulas with predicate symbols from P whose arguments are in the Herbrand universe of P. If the program contains a function symbol of nonzero arity, then the Herbrand universe and Herbrand base are countably infinite; otherwise they are finite. □

We shall be considering atoms in the Herbrand base and ground rules whose variables have been instantiated to elements of the Herbrand universe, which we call *instantiated* rules.

Definition 2.1.4: The *Herbrand instantiation* of a normal logic program is the set of rules obtained by substituting terms in the Herbrand universe for variables in every possible way. An *instantiated rule* is one in the Herbrand instantiation. Whereas "uninstantiated" logic programs are assumed to be a finite set of rules, instantiated logic programs may well be infinite. □

Definition 2.1.5: An Herbrand interpretation is an interpretation with the Herbrand universe as domain, with all terms in the Herbrand universe interpreted freely, with constants interpreted as "themselves" in the Herbrand universe, and with n-ary functions f interpreted as mappings from n-tuples $(t_1, \cdots, t_n)$ of terms from the Herbrand universe to the term $f(t_1, \cdots, t_n)$. □

We shall be dealing with both 2-valued Herbrand interpretations, which may be thought of as subsets of the Herbrand base, and 3-valued Herbrand interpretations, which can be thought of as consistent sets of literals whose atoms belong to the Herbrand base.

Definition 2.1.6: A *query* Q is a conjunction of literals. We use the notation $\leftarrow Q$ to denote a *goal*, so that resolution may be applied directly to the goal. A *ground substitution* for a goal G is a substitution of terms from the Herbrand universe of the program for all the variables in G. □

By abusing notation slightly we may write a query Q as $\{l_1, \cdots, l_n\}$ where each l_i is a literal. It should be understood that literals may be repeated, and so Q is a multiset rather than a set.

In the context of databases, it will be desirable to separate those predicates appearing only as facts, and those predicates appearing in the heads of rules with nonempty bodies.

Definition 2.1.7: Let q be a predicate appearing in the program P. q is an *extensional database* (EDB) predicate of P if q does not appear in the head of a rule with a nonempty body. q is an *intensional database* (IDB) predicate of P if q does appear in the head of a rule with a nonempty body. □

Definition 2.1.8: A normal logic program P is said to be *range restricted* if for every rule r in P, every variable occuring in either the head of r or in a negative literal in the body of r also occurs in a positive literal in the body of r. Such programs have also been called *allowed* [60] or *safe* [113]. □

Definition 2.1.9: 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. □

Definition 2.1.10: A program is *locally stratified* if there is an assignment of ordinal levels to *ground atoms* such that whenever a ground atom appears negatively in the body of an instantiated rule, the head of that rule is of strictly higher level, and whenever a ground atom appears positively in the body of an instantiated rule, the atom in the head has at least that level. □

In a database context we may want to reason about "all possible EDB's" without necessarily having available in the program all the symbols that would appear in such EDB's. To be able to reason in this way, we may assume that a universe $\mathcal{U}$ is given. In the case of function-free programs (and EDB's) we shall assume that $\mathcal{U}$ is finite; $\mathcal{U}$ should contain all constant symbols that can appear in all possible programs and EDB relations. In particular, $\mathcal{U}$ will include the Herbrand universe of any program/EDB pair. $\mathcal{U}$ will function as the domain under consideration, with terms interpreted freely. The choice of $\mathcal{U}$ can be problematic, in that in some cases it can affect the semantics of the program under consideration (see Section 3.13). For *range restricted* programs, the choice of $\mathcal{U}$ does not affect what is true according to their well-founded semantics.

Chapter 3

Declarative Semantics

3.1 Approaches to Semantics

There has been much recent work on providing a semantics for normal programs, both in the deductive database community and in the logic programming community. We give a brief overview of previous approaches in this section, and leave the formal definitions until later in this chapter.

3.1.1 Program Completion Semantics

The original "program completion" approach is due to Clark [27], and has been discussed in detail by Shepherdson [107, 108], Kunen [56], and Lloyd [60]. The idea behind the completed program is to replace collections of "if" definitions by one "if and only if" definition. The *completed program* (or *completed database*) is not a normal program, but a conjunction of "if and only if" formulas. The completed program is treated simply as a first order formula (see Section 3.4). Then literals that are logical consequences of the completed program, and only those, should be considered true. A proof method that supports this approach, called SLDNF-resolution (SLD-resolution plus the negation as failure rule) has been studied extensively.

Clark showed that SLDNF-resolution is sound in the sense that if a goal has a finite SLDNF derivation, then it is a logical consequence of the completed program. Jaffar, Lassez and Lloyd showed that SLDNF was complete (in the same sense) for *Horn* programs with non-floundering[1] queries consisting of a conjunction of positive and/or negative literals [48]. SLDNF-resolution was further investigated for *normal* logic programs by Lloyd [60] (who coined the term SLDNF-resolution), Shepherdson [107, 108] (*q.v.* for further bibliography), and others. For some programs the completed program is inconsistent; for some others, the completed program is consistent but gives too weak a semantics for database applications. We shall illustrate these claims with examples in Section 3.7.

Fitting [34] and Kunen [56] gave markedly different, more uniform, semantics by interpreting the completed program in a 3-valued constructive logic, elegantly eliminating some difficulties of the Clark program completion approach. The third truth value, $\perp$, connotes undefined truth value and is "less information than" both **true** and **false**, which are incomparable. Fitting

[1] A top-down method *flounders* if it encounters a negative subgoal containing an unbound variable. It is not sound to recursively expand a negative subgoal containing a variable since the quantification of the variable changes from universal to existential. See [60] for a discussion.

showed that the completion of every program has a (unique) minimum 3-valued model, and suggested that this model be taken for the semantics of the program. Kunen describes a variant that is always recursively enumerable, and characterizes the *3-valued logical consequences* of the completed program. See Section 3.4 for the formal definitions. From our point of view, however, these semantics are also too weak to capture the notion of negation as failure necessary in database applications, as discussed later in the motivating examples (Section 3.7).

3.1.2 Canonical Models

The "canonical model," or "preferred model," approach has been to declare that a certain model (or certain models) of the original program are presumed to be the "intended" ones, i.e., those that the programmer and users have in mind. The justification for choosing the preferred model relies on the expectations of a programmer or user, as discussed in Section 1.3. The advantage of assigning a canonical model to a program is that one now has a standard for correctness of a procedure for evaluating queries – it must conform to the canonical model, and succeed or fail appropriately.

Over the course of time, we may want to "apply" the same IDB to many different EDBs. In this context the properties of the IDB merit careful study, and it makes sense to think of the IDB as implicitly defining a transformation from an EDB to a set of derived facts; we would like the set of derived facts to be the canonical model. For finite cases the computational complexity of this transformation can be studied; see Section 3.9.

It can be shown that Horn programs have a unique least Herbrand model. This least model is well-accepted as a canonical model for Horn programs. A closely related idea, the *closed world assumption*, was introduced in the context of deductive databases by Reiter [91].

The *generalized closed world assumption* (GCWA) was proposed by Minker to handle disjunctive databases [70] without producing the inconsistency typical of the closed world assumption when the program is not Horn. The basic idea is to consider all *minimal* Herbrand models to be canonical ones. This semantics infers all sentences true in all minimal models, in a style similar to circumscription [67]. However, considering all minimal models ignores the implicit priority relationship between the head and the body of the rules. For example, the rule $p \leftarrow \neg q$ would be treated in an identical fashion to the rule $q \leftarrow \neg p$ by the GCWA.

In order to take into account such a priority relationship, researchers considered classes of programs for which there was an apparent ordering of the predicates into "levels" of priority. Two such classes are *stratified* programs (Definition 2.1.9) and *locally stratified* programs (Definition 2.1.10). The stratified class has been treated in [6, 24, 59, 116], and elsewhere. The locally stratified class, defined and studied by Przymusinski [82], is a superset of the class of stratified programs. Przymusinski defined *perfect* models, and showed that every locally stratified program has a unique perfect model. These classes are discussed further in Section 3.6.

For a while many researchers perceived programs that were not at least locally stratified as "not making sense," that they were inherently ambiguous, and thus faulty. Thus failure to have a perfect model was thought to indicate a flaw in the program rather than in the definition of perfect models. Recent experience has cast doubt on this attitude (see [37] for discussion), and spurred the search for further improvements in the definition of the "canonical model."

Gelfond and Lifschitz propose an elegant definition of a *stable model* that is closely related to our work [37]. Drawing on ideas in [36], they define a "stable model" as one that is able to reproduce itself in a certain sense; a program may have zero, one, or many stable models. In their scheme, when a program has a *unique* stable model, that is is considered to be its canonical model. They argue that the unique stable model is the natural one to associate with a program, and describe some of its properties. Stable models are discussed further in Section 3.5.

3.1.3 Well-Founded Semantics

In this thesis we propose a new definition of canonical model, which we call the *well-founded* model. We show that for locally stratified programs the perfect model coincides with the well-founded model; in addition, certain programs that are not locally stratified have a well-founded total[2] model. Examples are given in Section 3.7.

But even when a program has no well-founded *total* model, it has a well-founded *partial* model; thus we define the *well-founded semantics* of any normal program to be that ground literals in the well-founded partial model are true, their complements are false, and other literals' truth values are not determined by the program. Thus, a partial model can also be viewed as a model in 3-valued logic. This relationship is discussed in Section 3.4.

While stratification is a syntactic property of the IDB, for an unstratified IDB, whether the program has a total well-founded model depends in general on the EDB. One view of well-founded semantics is as an attempt to give a reasonable meaning to as much of the program as possible in the unfavorable cases, when only a partial model exists, as an extension of the semantics for the favorable cases, which have a total model.

The key idea in our formulation is the concept of an "unfounded set," which is an adaptation of the "closed set" developed for disjunctive databases by Ross and Topor [100], and is essentially the same as the "securable set" in [103]. Unfounded sets are defined in Section 3.3.

Since the well-founded semantics was presented at a conference [118], several alternative formulations of the well-founded semantics have been developed [18, 33, 84, 117]. We believe that this agreement indicates a robustness of the semantics, and provides evidence that it coincides well with intuition, at least in the context of deductive databases.

3.1.4 Inflationary Semantics

The inflationary semantics was proposed in [54] and, independently, in [3]. The idea behind the inflationary semantics is to treat negation as expressing "not *currently* true." If p is an atom, then $\neg p$ is considered satisfied if, *at the current stage of the iteration*, p has not been derived. It is of no concern if p is later established; no inferences are undone.

While first formally studied (in the absence of other features) in [3, 54], the inflationary interpretation of negation has been a feature of rule-based expert systems languages such as OPS5[3] [16, 29].

One may argue that the inflationary semantics does not provide a *declarative* semantics for negation on the grounds that the stage of the iteration is too important in defining a semantics.

[2] A model is *total* if it is 2-valued, i.e., nothing is made undefined. Since we are working within a 3-valued logic, we may also have *partial* models in which some atoms have the truth value "undefined."

[3] We don't consider languages such as OPS5 in this thesis. In OPS5 it is possible to update the database within rules, and so it certainly does not provide the sort of declarativeness we are looking for. Nevertheless, there may be a role for production-rule based languages like OPS5 in database systems, for "triggers" [122] or as a general purpose language.

For example, consider the program

$$
\begin{array}{l}
q \leftarrow t, \neg p \\
t \\
p
\end{array}
$$

At the first iteration t and p are inferred, and no more facts are generated at later stages. However, consider what happens if the atom t is deleted from the body of the first rule. One may hope that the semantics be unchanged since we shall immediately derive t anyway. However, after the deletion the first rule fires on the first iteration, since p is not inferred prior to the first iteration. Hence the semantics of the modified program includes the atom q.

Writing programs in such a language is likely to be more difficult due to the attention needed to be placed upon the stage of the iteration, and the resulting lack of modularity in the semantics of the language.

While the nature of inflationary programs is very different from the other approaches we consider in this thesis, we shall still be able to compare the inflationary semantics with other proposals on the issue of expressive power.

3.1.5 Formalisms in AI

Many of the issues involved with "reasoning by default" have been extensively studied within the Artificial Intelligence community, with the aim of modelling commonsense reasoning. The subfield dealing with negation by default has been known as "nonmonotonic reasoning."

Most of the AI formalisms are expressed in a logical framework, but aim to cover arbitrary theories rather than logic programs or deductive databases. Among these proposals are circumscription [67, 68], default theories [92] and autoepistemic logic [71]. These proposals are all based on classical logic: circumscription employs a second order minimization axiom; default theories use a "consistency" modal operator; autoepistemic logic uses a "belief" modal operator.

Przymusinski has shown an equivalence between the well-founded semantics and versions of each of these three formalisms when the AI formalisms are expressed in a 3-valued logic [86]. One must be cautious, however, since when expressed in 3-valued logic the modified formalisms yield significant differences from the corresponding 2-valued approaches.

3.2 Partial Interpretations

We shall be working extensively with sets of literals, for which we now introduce some notation. If p is an atom, then p is its *positive* literal, $\neg p$ is its *negative* literal, and these two literals are said to be *complements* of each other.

Definition 3.2.1: Let S be a set of literals. We denote the set formed by taking the complement of each literal in S by $\neg \cdot S$.

- We say that the literal q is *inconsistent with* S if $q \in \neg \cdot S$.
- Sets of literals R and S are *inconsistent* if some literal in R is inconsistent with S.
- A set of literals is *inconsistent* if it is inconsistent with itself; otherwise it is *consistent*.

The set of positive literals in S is denoted by $\mathbf{Pos}(S)$, and the set of negative literals by $\mathbf{Neg}(S)$. $\square$

Definition 3.2.2: Given a program P, a *partial interpretation* I for P is a consistent set of literals whose atoms are in the Herbrand base of P. A *total interpretation* is a partial interpretation that, for each atom p of the Herbrand base, contains either p or $\neg p$. We say a ground literal l is *true in* I when $l \in I$ and say l is *false in* I when the complement of l is in I. Similarly, we say a conjunction $l_1 \wedge \cdots \wedge l_n$ of ground literals is *true* in I if each l_i is true in I, and is *false* in I if some l_i is false in I. □

Definition 3.2.3: We say that an instantiated rule r is *satisfied* in a partial or total interpretation I if the head of r is true in I or some subgoal of r is false in I; r is *falsified* if the head of r is false in I and all subgoals of r are true in I. In addition, if the head of r is false in I, but no subgoal of r is false in I then we say that r is *weakly falsified* in I. □

Definition 3.2.4: A *total model* of a program P is a total interpretation such that every instantiated rule of P is satisfied. A *partial model* of P is a partial interpretation that can be extended to a total model of P. □

Although it is customary to omit the adjective "total" when speaking of interpretations and models, because we shall be dealing with both 2-valued and 3-valued logics, we shall include it in this chapter for clarity.

Intuitively, a partial interpretation may contain incomplete information: the positive literals in it are considered to be true atomic facts; the negative literals denote atoms considered to be false; and the truth values of the rest of the atomic facts are unknown, or unspecified, at least "at present." The natural ordering on partial interpretations is $\subseteq$: $I \subseteq I'$ if I' contains all the information in I, both positive and negative, plus possibly more.

For us, a partial model is a partial interpretation I such that some instantiated rules may not be satisfied, but there is a (possibly empty) set of literals whose addition to the partial interpretation yields a total model of the rules. Clearly, this is impossible if I falsifies any instantiated rule. If I only weakly falsifies some instantiated rule, then the addition of some *negative* literal to I may be necessary to satisfy that rule. Thus recognition of partial models containing weakly falsified rules may be difficult. The following lemma shows that the situation is much simpler if I does not weakly falsify any instantiated rule.

Lemma 3.2.1: Let P be a program and let I be a partial interpretation. If I weakly falsifies no instantiated rule from P, then I is a partial model of P.
Proof: Let I' be the total interpretation formed by adding to I all atoms in the Herbrand base that are neither true nor false in I. Let r be an instantiated rule from P. If I satisfies r, then so does I', since no instantiated rule is weakly falsified by I. If I does not satisfy r, then the head of r cannot be false in I, so it is true in I'. Hence I' is a total model. ∎

Our notion of partial model is not the same as the natural notions of models used in 3-valued logics, such as in the approaches of Fitting [34] and Kunen [56]. Nevertheless, the well-founded partial model we construct will also be a model in Fitting's 3-valued sense. We shall discuss 3-valued models in Section 3.4.

3.3 Unfounded Sets and Well-Founded Partial Models

In this section we define *unfounded sets*, which are a variation of *closed sets* that were defined for disjunctive databases by Ross and Topor in [100]. Unfounded sets provide the basis for negative conclusions in the well-founded semantics.

3.3.1 Unfounded Sets

Definition 3.3.1: Let a program P, its associated Herbrand base H, and a partial interpretation I be given. We say $A \subseteq H$ is an *unfounded set* (of P) *with respect to* I if each atom $p \in A$ satisfies the following condition: For each instantiated rule r of P whose head is p, (at least) one of the following holds:

1. Some (positive or negative) subgoal q in the body of r is false in I.

2. Some positive subgoal q in the body of r occurs in A.

A literal q that makes (1) or (2) above true is called a *witness of unusability* for rule r (with respect to I). □

Intuitively, we regard I as what we know *so far* about the intended model of P. Rules satisfying condition (1) are not usable for further derivations since their hypotheses are already known to be false.

Condition (2) is the unfoundedness condition: of all the rules that still might be usable to derive something in the set A, each requires an atom in A to be true. In other words, there is no one atom in A that can be *first* to be established as true by the rules of P ("knowing" I). Consequently, if we choose to infer that some or all atoms in A are false, there is no way we could later have to infer one to be true.

As described more formally later, the well-founded semantics uses conditions (1) and (2) to draw negative conclusions. Essentially, it simultaneously infers all atoms in A to be false. By contrast, the semantics of [34] uses only condition (1) to draw negative conclusions. The closed sets of Ross and Topor [100] were defined only with condition (2).

Example 3.3.1: Consider the program consisting of the eight (instantiated) rules below.

$$\begin{array}{l} p(a) \leftarrow p(c), \neg p(b) \\ p(b) \leftarrow \neg p(a) \\ p(e) \leftarrow \neg p(d) \\ p(c) \\[4pt] p(d) \leftarrow q(a), \neg q(b) \\ p(d) \leftarrow q(b), \neg q(c) \\ q(a) \leftarrow p(d) \\ q(b) \leftarrow q(a) \end{array}$$

The atoms $\{p(d), q(a), q(b), q(c)\}$ form an unfounded set with respect to $\emptyset$. In particular, $\{q(c)\}$ is unfounded due to Condition (1); there is no rule usable to establish its truth. The set $\{p(d), q(a), q(b)\}$ is unfounded due to Condition (2); we are given no way to establish $p(d)$ without first establishing $q(a)$ or establishing $q(b)$ (the status of $\neg q(b)$ in first rule for $p(d)$ is irrelevant for determining unfoundedness). Also, there is no way to establish $q(a)$ without first establishing $p(d)$, and no way to establish $q(b)$ without first establishing $q(a)$. Clearly $q(c)$ can never be proven, but we can also see that among $p(d)$, $q(a)$, and $q(b)$, none can be the *first* to be proven.

In contrast, the pair $\{p(a), p(b)\}$ does not form an unfounded set even though they depend on each other, because the only dependence is "through" negation. It is tempting to claim that the proof attempts for $p(a)$ and $p(b)$ will fail also, but such a claim is faulty.

The difference between sets $\{p(d), q(a), q(b)\}$ and $\{p(a), p(b)\}$ is this: Declaring any of $p(d)$, $q(a)$, or $q(b)$ false does not create a proof that any other element of the set is *true*. However, as soon as one of $p(a)$ or $p(b)$ is declared false, it becomes possible to prove the other is true. And if both are declared false at once, we have an inconsistency. □

Simultaneously negating all the atoms in an unfounded set effectively invokes the "closed-world assumption" [91] in Horn clause programs; if H is the Herbrand base and I is the set of atoms that represents the minimum Herbrand model of a Horn clause program [115], then $H - I$, the set of atoms not in I, is unfounded with respect to I. See Theorem 3.3.7.

It is immediate that the union of arbitrary unfounded sets is an unfounded set. This observation leads naturally to:

Definition 3.3.2: The *greatest unfounded set (of P) with respect to I*, denoted $\mathbf{U}_P(I)$, is the union of all sets that are unfounded with respect to I. □

We now make some observations about unfounded sets. To a certain extent, there is a flexibility between having $\neg p \in I$ and having p in an unfounded set. The following lemma shows that, given an interpretation R, if we deduce that certain facts S are in an unfounded set A and add their complements to R, other unfounded atoms remain unfounded.

Lemma 3.3.1: Let R be a set of literals, and let A be an unfounded set of P with repect to R. For any subset $S \subseteq A$, $A - S$ is unfounded with respect to $R \cup \neg \cdot S$.
Proof: Any witness of unusability that was an atom in S is now a negative literal in $\neg \cdot S$, and hence is still a witness. ▮

The next lemma demonstrates a connection between (lack of) weak falsification (Definition 3.2.3) and unfounded sets.

Lemma 3.3.2: Let I be a partial interpretation consisting of positive literals Q and negative literals $\neg \cdot S$. If I does not weakly falsify any instantiated rule of program P, then S is an unfounded set with respect to Q.
Proof: Let $p \in S$ and let r be any instantiated rule whose head is p. Because r is not weakly falsified, some subgoal of r is false in I. If this subgoal is positive, it is also in S, so condition (2) of Definition 3.3.1 is satisfied. If this subgoal is negative, its positive version is in Q so condition (1) is satisfied. ▮

3.3.2 Well-Founded Partial Models

We now consider a (possibly transfinite) sequence that results from combining two set transformations. The limit of this sequence defines the well-founded semantics. In what follows the word *transformation* always means a transformation between sets of literals, whose atoms are in the Herbrand base of a given program P. We recall that a transformation T is called *monotonic* if $T(I) \subseteq T(J)$, whenever $I \subseteq J$.

Definition 3.3.3: Transformations $\mathbf{T}_P$, $\mathbf{U}_P$, and $\mathbf{W}_P$ are defined as follows:

- $p \in \mathbf{T}_P(I)$ if and only if there is some instantiated rule r of P such that r has head p, and each subgoal literal in the body of r is true in I.

- $\mathbf{U}_P(I)$ is the greatest unfounded set of P with respect to I, as in Definition 3.3.2.
- $\mathbf{W}_P(I) = \mathbf{T}_P(I) \cup \neg \cdot \mathbf{U}_P(I)$. □

Lemma 3.3.3: $\mathbf{T}_P$, $\mathbf{U}_P$, and $\mathbf{W}_P$, are monotonic transformations.
Proof: Immediate from definitions. ▌

Unlike some other methods, our $\mathbf{T}_P$ treats positive and negative subgoals symmetrically. In deciding whether a negative subgoal $\neg p$ is true, some methods look for the absence of p from I. For us the absence of p is immaterial for the truth of the subgoal $\neg p$; we require the presence of $\neg p$.

Definition 3.3.4: Let α range over all countable ordinals. The sets I_α and I^∞ of ground literals are defined recursively by:

$$I_0 = \emptyset.$$

For successor ordinal $\alpha = \gamma + 1$,

$$I_{\gamma+1} = \mathbf{W}_P(I_\gamma).$$

For limit ordinal α,

$$I_\alpha = \bigcup_{\beta<\alpha} I_\beta.$$

Finally, define

$$I^\infty = \bigcup_\alpha I_\alpha.$$

Following Moschovakis [72], for any literal p in I^∞, we define the *stage* of p to be the least ordinal α such that $p \in I_\alpha$. We observe that the *stage* is always a successor ordinal for literals in I^∞. □

Lemma 3.3.4: I_α as defined in Definition 3.3.4 is a monotonic sequence of partial interpretations.
Proof: The proof is by induction on α. The basis, $\alpha = 0$, is immediate. For $\alpha > 0$, assume the lemma is true for $\beta < \alpha$.

For monotonicity, first let $\alpha = \gamma + 1$ be a successor ordinal. If the literal q is in I_γ, then there is a smallest $\beta < \gamma$ such that $q \in \mathbf{W}_P(I_\beta)$ (even if γ is a limit ordinal). But $\mathbf{W}_P$ is monotonic, so by the inductive hypothesis $q \in \mathbf{W}_P(I_\gamma)$. Monotonicity for limit α follows from the definition of I_α.

To show consistency for successor ordinal $\alpha = \gamma + 1$, note that every literal in I_α *first* appears in some $I_{\beta+1}$, i.e., at a successor ordinal "stage". Let A be any set of positive ground literals that has a nonempty intersection with (the positive literals of) $I_{\gamma+1}$. It is sufficient to show that A is not unfounded w.r.t. I_γ, for then the *greatest* unfounded set of I_γ is also disjoint from the positive part of $I_{\gamma+1}$ by the arbitrary choice of A. Choose the earliest $I_{\beta+1}$ that intersects A and select an atom p in that intersection. Then p was derived by some rule r all of whose subgoals are in I_β. By the inductive hypothesis, those subgoals are also in I_γ, and I_γ is consistent, so none of the subgoals is false in I_γ. By the choice of β, the positive subgoals are not in A. Thus rule r has no witness of unusability, which demonstrates that A is not an unfounded set w.r.t. I_γ.

For limit ordinal $\alpha > 0$, to show that I_α is a partial interpretation, assume the lemma is true for $\beta < \alpha$. If both q and $\neg q$ are in I_α, then there is some *successor* ordinal $\gamma + 1 < \alpha$ such that the same is true, contradicting the inductive hypothesis. ▮

It follows by classical results of Tarski that I^∞ is the least fixed point of the operator $\mathbf{W}_P$. The Herbrand base is countable, so for some countable ordinal α, $I^\infty = I_\alpha$.

Definition 3.3.5: The *closure ordinal* for the sequence I_α is the least ordinal α such that $I^\infty = I_\alpha$ (*cf.* [72]). □

Examples can be constructed where the closure ordinal is above ω, but such examples are likely to be very rare in practice. (See Example 4.3.1 in Chapter 4 for such a program.) In the case of a function-free program with a finite EDB, which is common in deductive databases, the limit is reached after a finite ordinal.

Definition 3.3.6: The *well-founded semantics* of a program P is the "meaning" represented by the least fixed point of $\mathbf{W}_P$, or the limit I^∞ described above; every positive literal denotes that its atom is true, every negative literal denotes that its atom is false, and missing atoms have truth value "undefined" ($\perp$) assigned by the semantics. We may sometimes refer to I^∞ as $M_{WF}(P)$. □

Lemma 3.3.5: Let I_α be as defined in Definition 3.3.4. Then I_α does not weakly falsify (Definition 3.2.3) any instantiated rule of P.
Proof: Let r be any instantiated rule with head p such that $\neg p \in I_\alpha$. We need to show that the body of r is false in I_α. By definition, $p \in \mathbf{U}_P(I_\beta)$ for some $\beta < \alpha$. By Lemma 3.3.4, $I_\beta \subseteq I_{\beta+1} \subseteq I_\alpha$. Either the body of r is false in I_β, or some subgoal q of the body of r is in the greatest unfounded set w.r.t. I_β. In the latter case, $\neg q \in I_{\beta+1}$, so the body of r is false in $I_{\beta+1}$. In either case, it follows that the body of r is false in I_α. ▮

Theorem 3.3.6: For each countable ordinal α, I_α in the sequence described in Definition 3.3.4 is a partial model of P.
Proof: Immediate by Lemmas 3.2.1 and 3.3.5. ▮

Definition 3.3.7: Suppose that for each p in the Herbrand Base I^∞ contains either p or $\neg p$, i.e. I^∞ is a total interpretation. Then by the above theorem, I^∞ is a total model, and we call this the *well-founded model*; otherwise we call I^∞ the *well-founded partial model*. □

Theorem 3.3.7: Every Horn program has a well-founded model I^∞, which is the minimum model in the sense of Van Emden and Kowalski [115], i.e., its positive literals are contained in every Herbrand model.
Proof: Let H be the Herbrand base and let Q be the set of positive literals of I^∞. Q is a fixed point of $\mathbf{T}_P$ [115]. In view of Theorem 3.3.6 it is sufficient to show that $H - Q \subseteq \mathbf{U}_P(I^\infty)$. Let p be any positive literal in $H - Q$. Each rule for p must have a positive subgoal that is also in $H - Q$, that subgoal being a witness of unusability for this rule. Thus $H - Q$ is unfounded with respect to $\emptyset$, and hence certainly is unfounded with respect to I^∞. ▮

3.4 Three-Valued Models of the Program Completion

The relationship of the well-founded semantics to other methods based on the program completion and 3-valued logics is discussed in this section. Clark introduced the completed program as a way of formalizing the notion that facts not inferable from the rules in the program were to be regarded as false [27]. Fitting studied models of the completed program in a 3-valued logic, and showed that all such models were fixed points of a certain operator [34]. We show that the well-founded partial model is also a model in this logic, but often not the least model.

3.4.1 The Program Completion

The idea behind the completion of a program is to collect all rules having the same head predicate into a single rule whose body is a disjunction of conjunctions, then replace the "if" symbol, "$\leftarrow$," by "$\leftrightarrow$." This states in effect that the predicate is completely defined by the given rules.

Definition 3.4.1: Let

$$p(t_1, \cdots, t_n) \leftarrow L_1, \cdots, L_m$$

be a normal rule, where each t_i is a term, and each L_j is a literal. Then the corresponding *transformed* rule is

$$p(X_1, \cdots, X_n) \leftarrow \exists Y_1 \cdots \exists Y_d((X_1 = t_1) \wedge \cdots \wedge (X_n = t_n) \wedge L_1 \wedge \cdots \wedge L_m)$$

where each X_i is a new variable, $Y_1, \ldots, Y_d$ are the variables in the original rule, and "=" is a distinguished (equality) predicate. Let p be a predicate and suppose there are $k \geq 1$ rules with head predicate p in a program P. Let

$$\begin{array}{c} p(X_1, \cdots, X_n) \leftarrow E_1 \\ \vdots \\ p(X_1, \cdots, X_n) \leftarrow E_k \end{array}$$

be the corresponding transformed rules. Then the *completed definition* of p in P is the formula

$$\forall X_1 \cdots \forall X_n (p(X_1, \cdots, X_n) \leftrightarrow E_1 \vee \cdots \vee E_k).$$

If there are no rules with head predicate p in P, then the *completed definition* of p in P is the formula

$$\forall X_1 \cdots \forall X_n \neg p(X_1, \cdots, X_n).$$

□

Definition 3.4.2: *Clark's equality theory* is given by the following axiom schema for the equality predicate "=".

1. $c \neq d$ for all pairs c and d of distinct constant symbols.
2. $\forall (f(X_1, \cdots, X_n) \neq g(Y_1, \cdots, Y_m))$ for all pairs f and g of distinct function symbols (of arities n and m respectively).
3. $\forall (f(X_1, \cdots, X_n) \neq c)$ for all constant symbols c and function symbols f (of arity n).

4. $\forall(t[X] \neq X)$ for each non-variable term $t[X]$ containing X.

5. $\forall(f(X_1, \cdots, X_n) = f(Y_1, \cdots, Y_n) \leftrightarrow (X_1 = Y_1) \wedge \cdots \wedge (X_n = Y_n))$ for each function f (of arity n).

6. $\forall(X = X)$.

7. $\forall((X_1 = Y_1) \wedge \cdots \wedge (X_n = Y_n) \rightarrow (p(X_1, \cdots, X_n) \rightarrow p(Y_1, \cdots, Y_n)))$ for each predicate p (of arity n) including $=$.

Here, $\forall F$ is the universal closure of the formula F. $\Box$

The equality axioms effectively force the equality predicate to interpret terms in the Herbrand universe freely.

Definition 3.4.3: The *completion* of a program P, denoted *comp*(P), is the collection of the completed definitions of all predicates in P together with Clark's equality theory. $\Box$

Example 3.4.1: Let P be the program

$$\begin{array}{l} p(a, X) \leftarrow q(X, Y), \neg r(Y) \\ p(X, Z) \leftarrow t(X, Y), p(Y, Z) \\ t(b, a) \\ q(d, e) \end{array}$$

Then the completion of P is

$$\begin{array}{l} \forall X_1 \forall X_2 [p(X_1, X_2) \leftrightarrow (\exists X \exists Y (X_1 = a) \wedge (X_2 = X) \wedge q(X, Y) \wedge \neg r(Y)) \vee \\ \qquad\qquad\qquad\qquad (\exists X \exists Y \exists Z (X_1 = X) \wedge (X_2 = Z) \wedge t(X, Y) \wedge p(Y, Z))] \\ \forall X_1 \forall X_2 [t(X_1, X_2) \leftrightarrow (X_1 = b) \wedge (X_2 = a)] \\ \forall X_1 \forall X_2 [q(X_1, X_2) \leftarrow (X_1 = d) \wedge (X_2 = e)] \\ \forall Y [\neg r(Y)] \end{array}$$

together with Clark's equality theory. $\Box$

Example 3.4.2: Recall the last four rules of Example 3.3.1, whose atoms formed an unfounded set:

$$\begin{array}{l} p(d) \leftarrow q(a), \neg q(b) \\ p(d) \leftarrow q(b), \neg q(c) \\ q(a) \leftarrow p(d) \\ q(b) \leftarrow q(a) \end{array}$$

The completion of this program is

$$\begin{array}{l} \forall X [p(X) \leftrightarrow ((X = d) \wedge q(a) \wedge \neg q(b)) \vee ((X = d) \wedge q(b) \wedge \neg q(c))] \\ \forall X [q(X) \leftrightarrow ((X = a) \wedge p(d)) \vee ((X = b) \wedge q(a))] \end{array}$$

together with Clark's equality theory. $\Box$

3.4.2 Three-Valued Models

The original "logical consequence" approach essentially declares that only conclusions that are logical consequences (in the classical, 2-valued sense) of the completed program should be inferred [27, 48, 60, 107]. When the completed program is consistent, this approach implicitly defines a 3-valued interpretation: assign value *true* to instantiated atoms that are true in all (2-valued, not necessarily Herbrand) models of the completed program, *false* to instantiated atoms that are false in all models, and $\perp$ (undefined) to all other instantiated atoms. However, because the truth of each literal is based on traditional 2-valued logic, we call this the *2-valued program completion* (2PC) interpretation.

The 3-valued interpretations were made explicit by Fitting [34] and Kunen [56], who also used 3-valued logic to evaluate formulas. Whereas $(p \vee \neg p)$ must be true in 2-valued logic, in 3-valued logic it may also be $\perp$. In addition, the "$\leftrightarrow$" produced by the program completion process was interpreted as Łukasiewicz's operator of "having the same truth value," so that $\perp \leftrightarrow \perp$ evaluates to true. Fitting's and Kunen's treatments eliminated some anomalies in the 2PC interpretation.

Example 3.4.3: Consider the single rule program

$$p \leftarrow \neg p, \neg q.$$

The Clark completion is

$$\begin{array}{l} p \leftrightarrow (\neg p \wedge \neg q) \\ \neg q \end{array}$$

which has no 2-valued model. However, if we add the "meaningless" rule,[4] evaluation $p \leftarrow p$, the completed program changes to:

$$\begin{array}{l} p \leftrightarrow (\neg p \wedge \neg q) \vee p \\ \neg q \end{array}$$

which has the unique 2-valued model, $\{p, \neg q\}$. If, instead, we add the "meaningless" rule, $q \leftarrow q$, the completed program changes to:

$$\begin{array}{l} p \leftrightarrow (\neg p \wedge \neg q) \\ q \leftrightarrow q \end{array}$$

which has a different 2-valued model, $\{\neg p, q\}$. However, all three versions have 3-valued models in which $p = \perp$. Finally, if we add several rules, giving:

$$\begin{array}{l} p \leftarrow \neg p, \neg q \\ q \leftarrow r \\ q \leftarrow s \\ r \leftarrow r \\ s \leftarrow s \end{array}$$

[4] This rule is meaningless in the sense that it is a logical tautology. From a "bottom-up" database approach it is also meaningless in the sense that p will not be derived using this rule unless p is already true. However, from a "top-down" logic-programming point of view, this rule introduces an infinite loop.

the completed program becomes:

$$\begin{aligned}
&p \leftrightarrow (\neg p \wedge \neg q)\\
&q \leftrightarrow (r \vee s)\\
&r \leftrightarrow r\\
&s \leftrightarrow s
\end{aligned}$$

Now there are three 2-valued models, which vary on whether r or s or both are true. Their common part (intersection) is $\{\neg p, q\}$. However, here the 2PC interpretation is *not a 3-valued model*. □

One principal result in [34] is that the completion of every program has a (unique) minimum 3-valued Herbrand model. Fitting suggests that this model be taken for the semantics of the program, and hereafter we call it the *Fitting model*. Thus the Fitting model is sometimes “less defined” than the 2PC interpretation, as in the previous example. However, Example 3.13.1 in Section 3.13 shows that the 2PC interpretation can be “less defined” than the Fitting model.

To any partial interpretation I there corresponds the obvious 3-valued interpretation in which atoms missing from I are assigned the truth value $\perp$. In this setting, our partial interpretations are the same as Fitting's basic sets [34]. In 3-valued logic literals and conjunctions are true and false in I as specified in Definition 3.2.2; in addition, the truth value $\perp$ may be assigned:

Definition 3.4.4: A ground literal q is called *undefined in I*, denoted by “$\perp$”, if neither q nor its complement is in I. A conjunction of ground literals evaluates to *undefined* in I if no literal in the conjunction is false in I and at least one is undefined in I. □

Definition 3.4.5: $\mathbf{N}_P$ is defined as the transformation that, given a 3-valued interpretation I, yields the set of atoms p such that for every rule r in the Herbrand instantiation of P with p as its head, the body of r is false in I, i.e., some subgoal of r is false in I. Note that $\mathbf{N}_P$ is the portion of $\mathbf{U}_P$ produced by condition (1) of Definition 3.3.1. □

Fitting also constructs 3-valued models with a fixed point operator [34]. For positive inferences, $\mathbf{T}_P$ is as in Definition 3.3.3. For negative inferences he uses (in effect) the transformation $\mathbf{N}_P(I)$ defined above. A second main theorem of that approach is:

Theorem 3.4.1: (Fitting) A 3-valued interpretation I is a 3-valued model of the completed program if and only if $I = \mathbf{T}_P(I) \cup \neg \cdot \mathbf{N}_P(I)$. ▮

This immediately yields a fixed point construction for 3-valued models, and the Fitting model is the least fixed point. We now show that the well-founded partial model is also a 3-valued model in Fitting's sense.

Theorem 3.4.2: Let I^∞ be as in Definition 3.3.4. Then $I^\infty = \mathbf{T}_P(I^\infty) \cup \neg \cdot \mathbf{N}_P(I^\infty)$. Hence, I^∞ is a 3-valued model of the completion of P.
Proof: Since $I^\infty = \mathbf{T}_P(I^\infty) \cup \neg \cdot \mathbf{U}_P(I^\infty)$, it follows that

1. $\mathbf{T}_P(I^\infty) \cup \neg \cdot \mathbf{N}_P(I^\infty) \subseteq I^\infty$, and
2. every positive literal in I^∞ is in $\mathbf{T}_P(I^\infty)$.

It remains to show that every negative literal $\neg p$ that is in I^∞ is also in $\neg \cdot \mathbf{N}_P(I^\infty)$. But by Lemma 3.3.5 each instantiated rule with head p has its body false in I^∞, so $p \in \mathbf{N}_P(I^\infty)$. That I^∞ is a 3-valued model of the completion of P follows from Theorem 3.4.1. ▮

Corollary 3.4.3: The Fitting model is a subset of I^∞. ∎

I^∞ can indeed differ from the smallest 3-valued model of the completion of the program, and need not even be a subset of all 2-valued models, as shown by the one-rule program, $p \leftarrow p$, in which p is false in I^∞ and is undefined in the Fitting model.

Kunen describes a variant that differs from Fitting's in two important ways: (1) the iteration is always stopped at ω, and (2) the Herbrand universe is defined with respect to a language with an infinite set of function symbols, which properly includes those that occur in the program [56]. The resulting 3-valued interpretation is recursively enumerable, but may not be a 3-valued model. Kunen's main theorem states that this interpretation characterizes the 3-valued logical consequences of the completed program.

3.5 Stable Models

Gelfond introduced an approach to negation through stable models [36], and motivated it by appealing to autoepistemic logic, as developed by Moore [71]. The theory has been further developed by Gelfond and Lifschitz [37], and also by Marek and Truszczynski [65, 66].

In this section we follow the definition of [37], which defines stability without reference to autoepistemic logic. We show that if a program has a total well-founded model, that model is the unique stable model. We also discuss two programs which do not have total well-founded models but do have unique stable models. Whether inferring (or not inferring) the truth of these extra literals is "a bug or a feature" of either approach we leave for the reader's judgement.

Gelfond and Lifschitz [37] define a stable model to be one that reproduces itself in a certain three stage transformation, which we call the *stability transformation*. If a program has only one stable model, that is called its unique stable model. Stable models refer to 2-valued logic. When speaking of total, or 2-valued, interpretations, it is more common to represent models as sets of ground atoms, with the understanding that missing atoms represent the negative literals. In this context a "minimal model" is one that has a minimal set of *positive* literals, and a "monotonic transformation" on total interpretations is one that is monotonic in terms of the positive literals alone. However, for consistency with the rest of this chapter, we shall represent models as sets of literals, and use the following notation for sets of positive and negative atoms in interpretations.

Definition 3.5.1: Given a normal logic program P, and its Herbrand instantiation, P_H, we define $\mathbf{S}$, the *stability transformation* from total interpretations into total interpretations. Given a total interpretation I, $\mathbf{S}(I)$ is defined by the following three stages:

1. Define
$$P' = T_1(P_H, I)$$
where T_1 is the following transformation: For each instantiated rule r, if r contains a *negative* subgoal that is inconsistent with I, then r is discarded. The output of the transformation is the set of instantiated rules that remain.

2. Define
$$P'' = T_2(P')$$
where T_2 is the transformation by which all negative subgoals are dropped from rules of P', leaving a Horn program. We call P'' the *reduction of P with respect to I*.

3. Since P'' is a Horn program, we can form its minimum (2-valued) model as in the standard least Herbrand model semantics [115]. In this context, "minimum," means that the set of positive literals is minimized, and hence the set of negative literals is maximized. We define $\mathbf{S}(I)$ to be this minimum model of P''. □

Example 3.5.1: Let P_H be

$$\begin{aligned} p &\leftarrow \neg p \\ a &\leftarrow \neg b \\ b &\leftarrow \neg a \end{aligned}$$

and let $M = \{a, \neg b, p\}$, which is a minimal model of P_H. Then P' consists only of

$$a \leftarrow \neg b.$$

because the other rules contain negative subgoals whose atoms are in $\mathbf{Pos}(M)$. Now P'' is the Horn rule

$$a.$$

Thus $\mathbf{S}(M) = \{a, \neg b, \neg p\}$, which, incidentally, is not a model of P_H. □

The name "stability transformation" is justified in a sense by the following lemma, which shows that $\mathbf{S}$ is a "shrinking" transformation (on positive literals) when applied to total models. However, as shown above, it is possible that M is a model and $\mathbf{S}(M)$ is not a model; it may "shrink" too much.

Lemma 3.5.1: Let M be a total model of a normal program P. Then

$$\mathbf{Pos}(\mathbf{S}(M)) \subseteq \mathbf{Pos}(M).$$

Proof: Using the terminology of Definition 3.5.1, M is a total model of P' and also of P'', by their construction. But $\mathbf{S}(M)$ is the *minimum* total model of P''. ▮

The models that are fixed points of $\mathbf{S}$ are of special interest.

Definition 3.5.2: A total model M of a normal program P is *stable* if it is a fixed point of $\mathbf{S}$; that is, if $M = \mathbf{S}(M)$. If program P has exactly one stable model, that model is called the *unique stable model* of P. □

It is immediate that a stable model is minimal (in terms of the set of positive literals), but not every minimal model is stable, as shown in Example 3.5.1 above and in Example 3.5.3 below.

Example 3.5.2: Let P_1 be

$$\begin{aligned} a &\leftarrow \neg b \\ b &\leftarrow \neg a \end{aligned}$$

Both $\{a, \neg b\}$ and $\{b, \neg a\}$ are stable models, so P_1 has no unique stable model. Its Fitting model, 2PC interpretation, and well-founded partial model are all empty. □

Example 3.5.3: Let P_2 be

$$p \leftarrow \neg p.$$

The only model of P_2 is $M = \{p\}$. The one rule in the program drops out of the reduction, making $\mathbf{S}(M) = \{\neg p\}$ the minimum model of the reduction of P_2. Hence P_2 has no stable model.

As discussed in Example 3.4.3, the completed program is $p \leftrightarrow \neg p$. Its 2PC interpretation in 2-valued logic is inconsistent. Its Fitting model and well-founded partial model are both empty. □

There is a close relationship between stable models and well-founded (partial or total) models. As defined, a unique stable model is demonstrated only through the explicit enumeration of all minimal models followed by testing each for stability. We shall show that well-founded total models are unique stable models. Theorem 3.5.2 below shows that being a fixed point of $\mathbf{S}$ is equivalent to being a fixed point of $\mathbf{W}_P$ for total models. In fact, this equivalence extends to all total interpretations because being a fixed point of either transformation ensures that the interpretation is a model. As shown in a later example, it is possible that a fixed point of $\mathbf{S}$ is not the *least* fixed point of $\mathbf{W}_P$, but if it is the least fixed point, that stable model is obviously unique.

Theorem 3.5.2: (Van Gelder, Schlipf) Let M be a total model of P. Then M is stable if and only if it is a fixed point of $\mathbf{W}_P$.
Proof: See [118]. ∎

Corollary 3.5.3: Let I be a total interpretation of P. Then I is a fixed point of $\mathbf{S}$ if and only if it is a fixed point of $\mathbf{W}_P$.
Proof: It is routine to show that if I is a fixed point of either $\mathbf{S}$ or $\mathbf{W}_P$, then every instantiated rule is satisfied. Hence I is a model, and Theorem 3.5.2 applies. ∎

Corollary 3.5.4: If P has a well-founded total model, then that model is the unique stable model. ∎

Corollary 3.5.5: The well-founded partial model of P is a subset of every stable model of P.
Proof: Every stable model is a fixed point of $\mathbf{W}_P$, and the well-founded partial model is the least fixed point. ∎

In Examples 3.5.4 and 3.5.5 below we show that the converse of Corollary 3.5.4 is not necessarily true.

3.5.1 Comparison of Stable and Well-Founded Approaches

We now compare the well-founded semantics with the stable model semantics. On many programs they yield identical results, and at first it appeared that the only difference was that the well-founded semantics defined a partial model when there were multiple stable models. However, it turns out that there also are programs with a unique stable model and only a partial well-founded model. In other words, the converse of Corollary 3.5.4 is not necessarily true. These examples and others show that awkward situations arise for well-founded models and unique stable models when the factoring operation of resolution theorem proving (or the law of the excluded middle, in natural deduction) plays a part. Recall that "factoring" of a ground clause is simply the operation of merging two identical literals.

Factoring enters the picture with a rule of the form

$$p \leftarrow \neg p, \ldots$$

because, as a disjunctive clause, it can be rewritten as

$$p \vee p \leftarrow \ldots$$

and then the two p literals can be merged. Another manifestation of this phenomenon occurs with a pair of rules,

$$\begin{array}{l} p \leftarrow a, \ldots \\ p \leftarrow \neg a, \ldots \end{array}$$

Again, as disjunctive clauses, they can be resolved on a, giving

$$p \vee p \leftarrow \ldots$$

and then the two p literals can be combined by factoring. Two-valued logical consequences that can be derived only by using factoring cannot be derived in either the well-founded semantics or the 3-valued program completion approaches (*cf.* Examples 3.4.3 and 3.5.3).

Example 3.5.4: Consider the program P_3 given by the four rules:

$$a \leftarrow \neg b \tag{1}$$

$$b \leftarrow \neg a \tag{2}$$

$$p \leftarrow \neg p \tag{3}$$

$$p \leftarrow \neg b \tag{4}$$

Let us first consider P'_3, consisting of just the first three rules above (*cf.* Example 3.5.1). The first two rules comprise P_1 of Example 3.5.2, which had two stable models; the third is P_2 of Example 3.5.3, which had no stable model. Thus the first three rules alone have two minimal models, neither of which is stable:

$$\{a, \neg b, p\} \qquad \text{and} \qquad \{\neg a, b, p\}$$

The program completion of P'_3 is inconsistent. (Just turn each "$\leftarrow$" into "$\leftrightarrow$".) Not too surprisingly, the well-founded partial model and the Fitting model are empty.

Adding the fourth rule would appear to be meaningless at first glance because p is already a (2-valued) logical consequence of the third rule, and there is no apparent basis to conclude $\neg b$, anyway. Nevertheless, the fourth rule has a strange effect: it stabilizes precisely one of the two models, and so produces a unique stable model for the full program! Moreover, the program completion of the full P_3,

$$\begin{array}{l} a \leftrightarrow \neg b \\ b \leftrightarrow \neg a \\ p \leftrightarrow (\neg p \vee \neg b) \end{array}$$

now has a 2-valued model. Whereas its well-founded partial model and Fitting model remain empty, the unique stable model of P_3 is

$$M = \{a, \neg b, p\}$$

The reduction of P_3 with respect to M is

$$\begin{array}{l} a \\ p \end{array}$$

This model is also the 2PC model. □

Example 3.5.5: Consider the program P_4 given by the four rules:

$$\begin{array}{l} a \leftarrow \neg b \\ b \leftarrow \neg a \\ c \leftarrow a,\, b \\ a \leftarrow \neg c \end{array}$$

Again the Fitting model and well-founded partial model are $\emptyset$, while the unique stable model exists and agrees with the 2PC model:

$$M = \{a, \neg b, \neg c\}$$

The reduction of P_4 with respect to M is simply

$$\begin{array}{l} a \\ c \leftarrow a,\, b \\ a \end{array}$$

□

3.6 Stratified and Locally Stratified Programs

Recall that a program is stratified (Definition 2.1.9) if one can assign levels to predicates so that the level of a predicate appearing negatively in the body of a rule is strictly lower than the predicate in the head, and so that the level of a predicate appearing positively in the body of a rule is of level no greater than the predicate in the head.

The *stratified semantics* of such a program is defined by first drawing all "level 0 inferences" in the normal way for Horn programs, and concluding $\neg p$ for all atoms p (with level 0 predicates) that have not been inferred. Note that this is not the usual "negation by failure" because some of these atoms may not have failed *finitely*; *cf.* Example 3.7.2. The definition of stratified semantics is completed inductively: After all atoms of levels less than k have been classified as positive or negative, use these literals to derive positive level k atoms; conclude $\neg q$ for all level k atoms q that have not been inferred. The result is called the *stratified model*.

It is immediate from Theorem 3.3.7 that the stratified semantics agrees with the well-founded semantics for level 0, and it is easy to see that the agreement extends to all levels. We shall prove a somewhat stronger result below.

Przymusinski carried the above idea to a finer grain by defining a program to be *locally stratified* (Definition 2.1.10) if it is possible to assign ordinal levels to ground atoms in the instantiated program with the same constraints on levels as for stratification [82]. Note that a locally stratified program is stratified if all atoms with the same predicate symbol can be assigned the same level. The extension handles situations where the "recursive negation" is apparent, but not real. A typical example is the program

$$\begin{array}{l} even(s(X)) \leftarrow \neg even(X) \\ even(0) \end{array}$$

where each ground atom can be given a level equal to the power of s in its argument.

To give a semantics to locally stratified programs Przymusinski [82] has given a definition for a *perfect model* of a program. We give a slightly different definition below.[5]

Definition 3.6.1: Given a level mapping for the ground atoms of a locally stratified program P, an Herbrand model M of P is a *perfect model* if for all other models M' of P the following condition holds: If l is the lowest level at which M and M' differ, then the restriction of M' to atoms from level l is *not* a subset of the restriction of M to atoms from level l. □

In other words the perfect model minimizes positive literals of lower level in preference to positive literals of greater level.

Przymusinski has shown that all locally stratified programs have a unique perfect model, and that it is independent of the level mapping chosen (within the constraints mentioned); moreover, on stratified programs, the perfect model agrees with the stratified model. We show that the well-founded semantics is an extension of this approach in the following sense.

Theorem 3.6.1: If P is locally stratified, then it has a well-founded model, which is identical to the (unique) perfect model.

Proof: We show by induction that for any atom p of level k: if p is in the perfect model, it is in the well-founded partial model I^∞; and if p is not in the perfect model, then $\neg p$ is in I^∞.

The basis, $k = 0$, is immediate.

For $k > 0$, first assume p is in the perfect model. Then we claim that there is an instantiated rule with p as head, say

$$p \leftarrow q_1, q_2, \ldots, \neg r_1, \neg r_2, \ldots$$

such that all q_i are in the perfect model and no r_j is in the perfect model. For if this were not so, we could remove p from the (supposedly) perfect model, and at worst have to add atoms of greater level than p (because they have a rule containing $\neg p$) to restore the model. Since the r_j are of lower level than p, the induction hypothesis asserts that $\neg r_j$ are in I^∞. Also, any q_i of lower level than p are in I^∞.

Now consider a program consisting of *all* instantiated rules with head atoms of level k, whose subgoals of lower level are true in I^∞. We modify the rules by removing the subgoals of level less than k, leaving a Horn program P'' (*cf.* Definition 3.5.2). Clearly the minimum model of P'' will be precisely the atoms of level k in the perfect model. But all such atoms are also in I^∞. Moreover, the atoms of level k not in the minimum model of P'' form an unfounded set of P'' with respect to $\emptyset$ by Theorem 3.3.7. It follows from the construction of P'' that these atoms also form an unfounded set of P with respect to I^∞, so their negations are in I^∞. ▮

3.7 Motivating Examples

In this section we compare well-founded semantics with some other recent approaches based on canonical models, the stable model semantics outlined in Section 3.5, and stratified semantics, which has been studied by many researchers. We present some examples to support our position that well-founded models are natural and intuitive, at least in certain contexts.

[5]Our definition assumes a level mapping is given, while Przymusinski's does not. Perfect models exist only when such a level mapping exists, or in the unusual situation in which a program has only one Herbrand model. We choose to ignore this second condition here. Also, the property of being a perfect model for a locally stratified program is independent of the particular level mapping used.

Example 3.7.1: This example is abstracted from the "Yale shootout" example due to Hanks and McDermott [43]. The program P is

$$\begin{aligned}
noise(T) &\leftarrow loaded(T),\ shoots(T) \\
loaded(0) & \\
loaded(T) &\leftarrow succ(S,T),\ loaded(S),\ \neg shoots(S) \\
shoots(T) &\leftarrow triggers(T) \\
triggers(1) & \\
succ(0,1) &
\end{aligned}$$

We regard *triggers* and *succ* as EDB predicates, and the others as IDB. The Herbrand instantiation of P contains ground versions of the IDB rules as follows:

$$\begin{aligned}
noise(1) &\leftarrow loaded(1),\ shoots(1) \\
noise(0) &\leftarrow loaded(0),\ shoots(0) \\
loaded(1) &\leftarrow succ(0,1),\ loaded(0),\ \neg shoots(0) \\
loaded(1) &\leftarrow succ(1,1),\ loaded(1),\ \neg shoots(1) \\
loaded(0) &\leftarrow succ(0,0),\ loaded(0),\ \neg shoots(0) \\
loaded(0) &\leftarrow succ(1,0),\ loaded(1),\ \neg shoots(1) \\
shoots(1) &\leftarrow triggers(1) \\
shoots(0) &\leftarrow triggers(0)
\end{aligned}$$

Intuitively, since we have no information that $shoots(0)$ holds, we are led to the (presumably) intended minimal model:

$$\begin{array}{l}
loaded(0), \neg shoots(0), \neg noise(0), \\
loaded(1),\quad shoots(1),\quad noise(1)
\end{array}$$

However, an alternative minimal model exists:

$$\begin{array}{r}
loaded(0), shoots(0),\ noise(0), \\
\neg loaded(1), shoots(1), \neg noise(1)
\end{array}$$

Since $noise(1)$ is not true in all minimal models, the circumscription approach does not allow it to be concluded, which was a main point made in [43]. However, the well-founded model is the intended one.

To compare with other approaches: The 2PC model and Fitting model are also the intended model here. The program is stratified, so the stratified semantics agrees with the well-founded semantics. The intended model is also the unique stable model, as the alternative is not stable. □

In the preceding example, the 2PC and Fitting models were 2-valued, and gave the intended model. The next example typifies the situation in which we consider the 2PC and Fitting models to be too weak an approach.

Example 3.7.2: Consider a program with the rules:

$$
\begin{aligned}
tb(X,Y) &\leftarrow b(X,Y)\\
tb(X,Y) &\leftarrow b(X,U),\ tb(U,Y)\\[4pt]
tg(X,Y) &\leftarrow g(X,Y)\\
tg(X,Y) &\leftarrow g(X,U),\ tg(U,Y)\\[4pt]
a(X,Y) &\leftarrow tg(X,Y),\ \neg tb(X,Y)
\end{aligned}
$$

and the facts about b and g:

$$
\begin{array}{ll}
b(1,2) & g(2,3)\\
b(2,1) & g(3,2)
\end{array}
$$

Apparently, tb is the transitive closure of b and tg is the transitive closure of g. We expect a to be the difference of these two relations; in particular, it seems that $a(2,3)$ is true. This appears to be the intended model, and is indeed the well-founded model, as well as the stratified model.

There is another minimal model, in which $tb(2,3)$ and $tb(1,3)$ are true and $a(2,3)$ is false. Moreover, this alternative model satisfies the Clark completion of the program as well. Thus by the method of logical consequences of the completion of the program, the status of $a(2,3)$ and other literals is either not addressed (2PC interpretation) or declared undefined (Fitting model, Kunen model).

The criterion of stability reinforces the choice of the well-founded model, which is the unique stable model. □

In fact, Kunen has recently shown that in his 3-valued logical consequence semantics, a "strict" logic program without function symbols cannot define a predicate that is true in the transitive closure of a relation, false in its complement, and nowhere undefined [57]. Informally, a "strict" program is one in which the dependence of one predicate on another (or itself) is either through an even number of negations or through an odd number, but not both.

As another motivational example, we consider a program that is not locally stratified, as defined in Section 3.6, yet has a well-founded model when the EDB relation is acyclic.

Example 3.7.3: This example is essentially the same as one discussed by Gelfond and Lifschitz [37], and is one of the examples that led to the formulation of well-founded semantics, as well as the stable model semantics. Interestingly, this program turns out to be closely related to a game described by Kolaitis, and used to prove that there are queries in fixpoint logic that are not expressible by function-free stratified programs [53]. The program can be viewed as describing a game where one wins if the opponent has no legal moves, as in checkers (draughts).

$$winning(X) \leftarrow move(X,Y),\ \neg winning(Y)$$

Whenever the *move* EDB relation is acyclic the well-founded total model is easily found, by proceeding "up" the directed graph. If the *move* relation consisted of the tuples $move(a,b)$ and $move(b,a)$ then the well-founded model would be partial, making both $winning(a)$ and $winning(b)$ undefined. However, even when a cycle is present in the EDB, there may be a well-founded total model. For example, if the *move* relation contained the facts $move(a,b)$, $move(b,a)$ and $move(a,c)$ then the well-founded model would be total, containing $winning(a)$, $\neg winning(b)$ and $\neg winning(c)$. For this program, the Fitting model and the 2PC interpretation agree with the well-founded model.

However, the program is not locally stratified because the Herbrand instantiation contains a rule in which *winning* depends negatively upon itself, as in

$$winning(a) \leftarrow move(a, a), \neg winning(a).$$

This also destroys the perfect model[6] even though $move(a, a)$ does not occur in the EDB. We shall generalize the concept of local stratification to "modular stratification" in Chapter 5; when the *move* relation is acyclic, the above program is modularly stratified. □

For an extended example involving a VLSI layout problem that might easily arise in practice, see [118].

A theme that runs through these examples is that well-founded semantics frequently agrees with other semantics, but seems to avoid their awkward cases. In this sense it seems robust.

3.8 Compositionality

As discussed in Section 1.4, compositionality is an important requirement of a deductive database system, and a prerequisite for modularity. We shall say that a semantics is *compositional* if the following condition holds. This condition is based on Schlipf's "Principle of Stratification" [104].

Definition 3.8.1: (Compositionality) Suppose P_1 and P_2 are programs such that, for some set of predicates T, P_1 has predicates from T in the body of its rules, but not in the head. (In other words, the predicates in T are *used by* P_1 but not *defined in* P_1.) Suppose that some of the predicates in T are defined in P_2, and that apart from T, P_1 and P_2 share no predicate symbols. Then we say a semantics S is *compositional* with respect to P_1 if for all P_2 assigned a semantics according to S, that semantics extends to a semantics of $P_1 \cup P_2$. We say a semantics is *compositional* if it is compositional with respect to every such P_1. □

Both the well-founded semantics and Fitting's semantics are compositional in this sense. The 3-valued model of lower-component predicates can be extended to a model of the whole program simply by iterating $\mathbf{W}_P$ or $\mathbf{N}_P$ respectively. Also, the stratified semantics is compositional (for stratified programs). However, neither the 2PC interpretation nor the stable model semantics are compositional, as illustrated in the following example.

Example 3.8.1: Let P_1 be the program

$$p \leftarrow q, \neg p$$

and let P_2 be the simple program

$$q.$$

Let $T = \{q\}$, where T is as in Definition 3.8.1. Then both the 2PC interpretation and the stable model semantics assign P_2 the unique model $\{q\}$. However, the completion of $P_1 \cup P_2$ has no 2-valued models, and there are no stable models of $P_1 \cup P_2$. □

[6]Except in the trivial case where the program has only one Herbrand model.

There are weaker notions of compositionality that are also useful. Consider a semantics S that gives the well-founded model if 2-valued, but is undefined if the well-founded semantics is 3-valued. In general S is not compositional. However, consider the game example (Example 3.7.3), and let P_1 be the single rule

$$winning(X) \leftarrow move(X, Y), \neg winning(Y).$$

Then S is compositional with respect to P_1 if we restrict P_2 to have a semantics in which the *move* relation is acyclic. Such a restriction is natural for some examples; the semantics of the parts-explosion program presented in Section 5.6 is two-valued if the part-subpart relationship is acyclic, something one would expect of a part hierarchy.

3.9 Computational Complexity

Not only do we want to formulate a reasonable semantics for negation, we also want the set of statements derivable to be "reasonably computable," as far as possible. Unfortunately, the well-founded partial model is not necessarily recursively enumerable, a difficulty it shares with most of the semantics discussed here. However, for function-free programs, the Herbrand universe is finite and the construction is effective. Theorem 3.9.1 below shows that the *data complexity* of the well-founded semantics, as defined by Vardi [119], is polynomial. From this standpoint it is competitive with other methods, such as the stratified semantics, whose data complexity has been studied elsewhere [23, 42, 47, 119], and the Fitting model (as remarked below).

Definition 3.9.1: The *data complexity* of an IDB with respect to a fixed ground atomic query Q is defined as the computational complexity of deciding the answer to Q as a function of the size of the EDB. In the context of well-founded semantics, "deciding the answer" means deciding whether the ground atom is positive in the well-founded partial model. We say that the data complexity of an IDB belongs to a complexity class C if, for every fixed ground atomic query Q, the data complexity of the IDB with respect to Q is in C. □

Theorem 3.9.1: (Schlipf, Vardi) The data complexity of the well-founded semantics for function-free programs is polynomial time.
Proof: See [118]. ▮

The Fitting model also has polynomial data complexity (for function-free programs). The proof is similar to that of Theorem 3.9.1 above.

In contrast, Marek and Truszczynski [65] have shown that, even for *propositional* normal logic programs P, determining whether P has a stable model at all is NP-complete.

3.10 Expressive Power

In this section we compare the various semantics according to the (formal) criterion of expressiveness. There are several types of expressiveness results.

One important variety is of the type "Semantics X over the class of programs Y cannot express the query Q." If we can show such results for queries Q that we need to be able to ask, then we can conclude that the semantics is inadequate.

Another important type is "Over the class of programs Y, semantics X can express every query expressible in semantics Z." This type of result indicates that one semantics is at least as expressive as the other. Proving the converse of such a result would demonstrate the equivalence of the two semantics' expressive powers.

A third type is "Over the class of programs Y, semantics X expresses exactly those queries computable in time complexity C." Results linking complexity-classes with definability are interesting because they give a computational characterization of what a semantics can define.

So far, we have not really specified what it means to "express" a query. We shall do so below. We also remark that expressiveness may depend not only on the semantics and on the class of programs but also on the interpretations that are considered. For example, it is reasonable in the context of function-free databases to consider only *finite* interpretations. Sentences that are true in all finite models are not necessarily true in all models, and so more queries will be expressible over finite interpretations than over all interpretations.

Let us suppose that a semantics S for a program P can be expressed as a 3-valued interpretation. All semantics we consider in this section can be expressed in this form. For the well-founded semantics, the Fitting semantics, the inflationary semantics and the stratified semantics the 3-valued interpretation is simply the least fixpoint of the appropriate construction. For the stable model semantics we may say that a ground literal l is true with respect to the stable model semantics if l belongs to all stable models.[7]

The following definition is based on [104]. If S is a semantics, then let $S_{P,D}$ be the corresponding 3-valued interpretation for the IDB P and EDB D.

Definition 3.10.1: A relation $s(D)$ on EDB's D is *definable* in a semantics S over a class of (IDB) programs $\mathcal{P}$ if there is a program $P \in \mathcal{P}$ and a relation r in P such that either

$$\forall D : s(D) = \{\vec{X} : r(\vec{X}) \in S_{P,D}\} \quad \text{or} \quad \forall D : s(D) = \{\vec{X} : \neg r(\vec{X}) \in S_{P,D}\}.$$

□

Given a query Q, we may write $Q(D)$ to denote the answer to the query Q for a particular EDB D. We say that the query Q is *expressible* in a semantics S if the relation $Q(D)$ is definable in S.

The following theorem summarizes results from [3, 53, 54, 104].[8] Here, "$\subset$" denotes proper containment. Also, Π^1_1 denotes the class of queries expressible by second-order formulas of the form

$$\forall X_1 \cdots \forall X_n \; \phi(\vec{X})$$

where ϕ is first-order, and the variables X_i are second order, ranging over relations rather than values.

Theorem 3.10.1: For function-free programs over finite EDB's the following relationships hold

[7] If the program has no stable models then for the purposes of this section on expressive power we shall follow the approach of [104] an include every literal in the "interpretation." We shall ignore the fact that the resulting structure is inconsistent (and hence not really a 3-valued interpretation).

[8] These results make assumptions about the existence of certain distinguished constant symbols. See the cited references for these minor assumptions.

between queries expressible in the following semantics:

$$\begin{array}{c} \text{Datalog} \\ \subset \\ \text{Stratified} \\ \subset \\ \text{Fitting} = \text{Well-founded} = \text{Inductive} = \text{Inflationary} \\ \subset \\ \mathbf{P} \\ \subseteq \\ \text{Stable} = \textbf{co-NP} = \Pi^1_1 \end{array}$$

▮

Here "Datalog" represents the least model semantics for negation-free programs and "Stratified" represents the stratified semantics for stratified programs. "Inductive" represents the class of inductively definable queries [72], which are those queries definable in first-order fixpoint logic.

The following theorem summarizes results from [104].

Theorem 3.10.2: For normal programs over EDB's giving infinite Herbrand universes, the following relationships hold between queries expressible in the following semantics:

$$\text{Fitting} = \text{Well-founded} = \text{Inductive} = \text{Stable} = \Pi^1_1$$

▮

There are several other interesting expressibility results in the literature. By assuming a distinguished predicate that provides an ordering of database elements, it can be shown that on finite EDB's the inductively definable relations are precisely those computable in polynomial time [23, 42, 47, 119].

Schlipf has shown that while equally expressive over infinite Herbrand universes, the translation from Fitting's semantics to the well-founded semantics cannot be performed in a *compositional* manner [104]. If one makes compositionality a requirement of the simulation of one semantics by another, then the well-founded semantics is more expressive than Fitting's semantics.

3.11 A Comparison of Semantics

In this section we summarize the material in this chapter by comparing the various different semantics according to the criteria developed in Section 1.4. Table 3.1 summarizes the results for normal programs, and Table 3.2 summarizes the results for function-free programs. A "?" indicates that the particular result is (as far as the author knows) unknown.

	Stratified semantics	Fitting semantics	Well-Founded semantics	Stable model semantics
Decidable	No	No	No	No
Appropriate for	databases	logic programming	databases	databases
Applicable to	(locally) stratified programs	all programs	all programs	some programs[9]
Expressive power on infinite Herbrand structures	?	inductively definable	inductively definable	inductively definable
Compositional	Yes	Yes	Yes	No

Table 3.1: Comparison of declarative semantics for normal programs.

	Stratified/Perfect model semantics	Fitting semantics	Well-Founded semantics	Stable model semantics
Decidable	Yes	Yes	Yes	Yes
Data Complexity	Polynomial	Polynomial	Polynomial	NP-complete
Expressive power on finite structures	Less than inductively definable	inductively definable	inductively definable	co-NP

Table 3.2: Comparison of declarative semantics for function-free programs.

3.12 Beyond the Well-Founded Semantics

The major shortcoming of the well-founded semantics that we have found concerns its inability to handle conclusions that can be reached only by using factoring or a similar technique, such as "ancestor resolution." Such techniques are known to be necessary for completeness of non-Horn proof systems, but not for sets of Horn clauses. The need for factoring arises principally from "proof by cases", and sometimes from "proof by contradiction".

[9]All stratified programs, and all programs with two-valued well-founded models have a unique stable model. In general, though, programs may have no stable model at all.

However, factoring possibilities in the given program do not always carry over to the completed program, and $a \vee \neg a$ does not simplify to **true** in either 3-valued logic [34, 56] or intuitionistic logic [32]. Thus caution is needed to keep a coherent system.

The overly trivial P_2 in Example 3.5.3 might lead one to believe that a factoring capability can be easily "patched in" by just checking for a negative subgoal that complements the head of the rule; this conclusion would be incorrect, as shown by

$$\begin{aligned} a &\leftarrow \neg b \\ b &\leftarrow \neg a \\ p &\leftarrow a \\ p &\leftarrow b \end{aligned}$$

in which we cannot choose between a and b, but might reasonably be expected to notice that p must hold (in 2-valued logic). In general, recognizing that p is a (2-valued) logical consequence of a *finite* set of instantiated rules is co-NP-complete. Furthermore, we normally start with rules that contain variables. Thus, any extension of logic program semantics that depends on "true non-Horn reasoning" needs to be undertaken with great caution, and represents a significant open problem.

3.13 Augmented Programs

Certain programs exhibit undesirable behavior when interpreted in the Herbrand universe, due to their containing what is called *unsafe negation*. A simple way to remove this behavior is to augment the program, as described in this section. We proceed informally here, and refer to [63] for a formal discussion.

Definition 3.13.1: Any normal logic program P has an associated *augmented program* that is formed by adding the rule

$$\bar{p}(\bar{f}(\bar{c})) \leftarrow \bar{p}(\bar{f}(\bar{c})).$$

where $\bar{p}$, $\bar{f}$, and $\bar{c}$ are symbols that do not occur elsewhere in the program. □

The extra terms in the augmented Herbrand universe provide infinitely many additional terms that were not in the original Herbrand universe. The extra terms ensure that goals with free variables have "room to fail" when they should, even in their instantiated versions. Augmenting achieves an effect similar to Kunen's embedding the program in a language with infinitely many function and constant symbols.

Example 3.13.1: In the following program, without inspecting the a relation, we would expect p to hold wherever a does. (Read d as "differs" and s as "same".)

$$\begin{aligned} p(X) &\leftarrow a(X),\, d(X,Y) \\ d(X,Y) &\leftarrow \neg s(X,Y) \\ s(U,U)& \\ a(1)& \end{aligned}$$

The underlying idea is that, looking at the rule for s, we expect the formula $\forall Y\, s(X,Y)$ to be false. But in the unaugmented Herbrand universe of one element there is no "room" for $s(1,Y)$

to fail because 1 is the only term. As a result, $p(1)$ fails. However, adding the apparently unrelated fact $b(2)$ to the program means that $s(1, Y)$ can fail, by setting $Y = 2$. The failure of $s(1, 2)$ provides a proof for $d(1, 2)$, allowing a proof of $p(1)$. Augmenting the program avoids this bizarre behavior; $s(1, \bar{c})$ fails in all cases, making $p(1)$ always provable, as expected.

To see why this program has unsafe negation, consider a top-down sequence of goal reductions beginning with $p(1)$. Using the rules, $p(1)$ reduces to $(a(1), d(1, Y))$, $a(1)$ reduces to true, then $d(1, Y)$ reduces to $\neg s(1, Y)$. The occurrence of a free variable Y in the negative subgoal is called "unsafe" because it is not limited to any domain. This derivation is said to have floundered [60].

Finally, let us note that in the unaugmented program $p(1)$ is false in the well-founded semantics and in the Fitting semantics, but not in the 2PC semantics or Kunen semantics. Although $p(1)$ is false in the only 2-valued *Herbrand* model of the completed program, there are other 2-valued models in which $p(1)$ is true. All of these semantics agree that $p(1)$ is true in the augmented program. □

Przymusinski proposes an alternative solution to what he terms the "Universal Query Problem." In proving the soundness and completeness of SLS-resolution with respect to the perfect model semantics, he considers *all* perfect models rather than just perfect Herbrand models [85]. These models have to be models of Clark's equality theory (Definition 3.4.2).

Chapter 4

Procedural Semantics

In this chapter we shall look for a procedural counterpart to the declarative semantics of Chapter 3. First, we briefly examine past work on procedural semantics.

4.1 Previous Approaches

Based on the completion, Clark proposed a top-down procedural semantics known as "Negation as Failure," which when combined with SLD-resolution [115] is referred to as SLDNF-resolution. This method is sound with respect to the completion of the program, and is complete for Horn programs (possibly with negative subgoals in the goal only) [48].

Based on the perfect model approach, Przymusinski introduced SLS-resolution [85]. SLS-resolution is a top-down procedural semantics that uses an extension of SLD-resolution to answer queries. Przymusinski showed that for stratified programs with non-floundering queries, SLS-resolution is sound and complete with respect to the unique perfect model of the program. Unfortunately, SLS-resolution is not effective in general. It was argued in [85] that SLS-resolution may be considered a theoretical construct, an ideal query answering procedure to which various effective approximations may be compared. An alternative justification is that SLS-resolution is effective for a limited class of programs, and for a less limited class (say, the class of stratified function-free programs) SLS-resolution can be extended with memoing features to make it effective [51, 106]. (See Chapter 5 for further discussion of memoing.)

The main drawback of SLDNF-resolution is that it is too weak. SLDNF-resolution suffers from the same drawbacks as the completion discussed in Chapter 3. (See Example 3.7.2.) The main drawback of SLS-resolution is that it is well-defined only for stratified or locally stratified programs.

In this chapter we present a top-down procedural "implementation" of the well-founded semantics. We call this procedure, which extends SLS-resolution, "global SLS-resolution." We show that for arbitrary programs, and for non-floundering queries, global SLS-resolution is sound and complete with respect to the well-founded model of the (augmented) program. (What we refer to as completeness some authors call "partial completeness" to indicate possible non-termination.) As with SLS-resolution, global SLS-resolution is not effective in general. We cannot expect to have an effective procedure as the well-founded partial model is not recursively enumerable in general, even when restricted to non-floundering literals. Nevertheless, for subclasses of programs we can guarantee termination, and we can also extend global SLS-resolution with memoing features for larger classes of programs (see Chapter 5). For further discussion see Section 4.7.

Przymusinski had independently described a similar procedural semantics that he also calls "SLS-resolution" [83].

4.2 Terminology

Definition 4.2.1: A *computation rule* is a rule for selecting one or more literals from a query. We assume that a computation rule may depend more than the current query. Computation rules are also referred to as *selection functions.* □

Definition 4.2.2: ($\models$) For ground queries $Q = \{p_1, \ldots, p_n, \neg q_1, \ldots, \neg q_m\}$, we shall write $M_{WF}(P) \models Q$ if and only if $Q \subseteq M_{WF}(P)$. Also $M_{WF}(P) \models \neg Q$ (the negation of the conjunction of elements of Q) if and only if either some q_j or some $\neg p_i$ is in $M_{WF}(P)$.

We write $M_{WF}(P) \models \forall Q$, where Q is a query or negated query, possibly containing variables, to mean $M_{WF}(P) \models Q\alpha$ for every ground substitution α of terms from the Herbrand universe for the variables of Q. We write $M_{WF}(P) \models \exists Q$, to mean $M_{WF}(P) \models Q\alpha$ for some such ground substitution α. □

We now give a slightly different definition of the well-founded partial model than that of Section 3.3.2. The operators $\mathbf{T}_P$ and $\mathbf{U}_P$ are from Section 3.3.2. The alternative definition below has technical advantages for the proofs of the results in this chapter. Define

$$\bar{\mathbf{T}}_P(I) = \mathbf{T}_P(I) \cup I.$$

Since $\mathbf{T}$ is monotonic (and in fact continuous), so is $\bar{\mathbf{T}}$. Let $\mathbf{V}_P$ be defined by

$$\mathbf{V}_P(I) = (\bigcup_{k=1}^{\infty} \bar{\mathbf{T}}_P^k(I)) \cup \neg \cdot \mathbf{U}_P(I).$$

$\mathbf{V}_P$ is monotonic, and has a least fixpoint. We can construct the least fixpoint of $\mathbf{V}_P$ by the following transfinite iteration.

Definition 4.2.3: Let α and β be countable ordinals. The partial interpretations I_α and I^∞ are defined recursively by

$$I_0 = \emptyset.$$

For successor ordinal $\alpha + 1$,

$$I_{\alpha+1} = \mathbf{V}_P(I_\alpha).$$

For limit ordinal α,

$$I_\alpha = \bigcup_{\beta<\alpha} I_\beta.$$

Finally, define

$$I^\infty = \bigcup_{\alpha} I_\alpha.$$

For any literal p in I^∞, we define the *stage* of p (written $stage_p$) to be the least ordinal α such that $p \in I_\alpha$. The above definition implies that the stage is always a successor ordinal for literals in I^∞. □

It is straightforward to show that the sequence of partial interpretations I_α is monotonically increasing, i.e., $I_\beta \subseteq I_\alpha$ if $\beta \leq \alpha$.

Lemma 4.2.1: I^∞ is the least fixpoint of $\mathbf{V}_P$, and is equal to $M_{WF}(P)$.
Proof: That I^∞ is the least fixpoint of $\mathbf{V}_P$ is a consequence of classical fixpoint results of Tarski for monotonic operators over complete lattices. It is clear by the definitions of $\mathbf{V}_P$ and $\mathbf{W}_P$ that for every partial interpretation I, $\mathbf{W}_P(I) \subseteq \mathbf{V}_P(I)$, and hence $M_{WF}(P) \subseteq I^\infty$.

We now define a sequence of partial interpretations that is similar to the sequence of Definition 4.2.3, except that we iterate $\mathbf{W}_P$ rather than $\mathbf{V}_P$. Let α and β be countable ordinals. The sets I'_α of partial interpretations are defined recursively as in Definition 3.3.4; $M_{WF}(P)$ is the least upper bound of the interpretations I'_α.

From the definitions of I_α and I'_α, it is easily shown by induction that $I_\alpha \subseteq I'_{\omega\alpha}$, where $\omega\alpha$ is the ordinal product of ω and α. Since $\bar{\mathbf{T}}$ is continuous, it has closure ordinal ω. Hence $I^\infty \subseteq M_{WF}(P)$, and the result follows. ▮

4.3 Global Trees and Global SLS-Resolution

In this section we define SLP-trees, which form the basis of the definition of Global Trees. These in turn form the basis of the definition of global SLS-resolution.

Definition 4.3.1: Let R be a computation rule. We say that R is *safe* if it never selects a non-ground negative literal from a query. We say that R is *positivistic* if it selects positive literals ahead of negative ones. R is *negatively parallel* if given a query containing only negative literals, it selects all (and only) the ground negative literals appearing in the query. The positivistic and negatively parallel conditions imply safety; a positivistic and negatively parallel rule is said to be *preferential.* □

In order to achieve soundness in a derivation, we require the computation rule to be safe. As we shall see, in order to achieve completeness of global SLS-resolution with respect to the well-founded semantics we will require a positivistic and negatively parallel computation rule. We now define SLP-trees. The "SLP" stands for "Linear resolution using a Positivistic Selection function."

Definition 4.3.2: (SLP-trees) Let G be the goal $\leftarrow Q$ and let R be a positivistic computation rule. We define the SLP-tree T_G for G. The root node of T_G is G. If the goal $H = \leftarrow Q'$ is any node of T_G then its children are obtained as follows:

- If Q' contains a positive literal then the literal L selected by R from Q' must be positive. Let U_L be the set of program clauses whose heads unify with L. The children of H are obtained by resolving H with (a variant of) each of the clauses in U_L over the literal L using most general unifiers. If there is no such K then H has no children, and is a *dead* leaf.
- If Q' is empty, or contains only negative subgoals then Q' is an *active* leaf.

A *branch* of T_G is an acyclic path from the root of T_G. We associate with each active leaf L its *computed most general unifier*, which is the composition of the most general unifiers used along the branch to L. $\square$

See Example 4.3.1 for examples of SLP-trees. We now define the global tree for a goal in terms of SLP-trees. The global tree may be thought of as an "OR/NOR" tree in the style of "AND/OR" trees.

Definition 4.3.3: We define the *global tree* Γ_G for a goal G. The nodes of Γ_G are of three types: *negation nodes*, *tree nodes*, and *non-ground nodes*. Tree nodes are actually SLP-trees for intermediate goals. The root of Γ_G is the SLP-tree for the goal G. An *internal* tree node is a tree node that is not the root.

Let T_H be any tree node of Γ_G. The children of T_H are negation nodes, one corresponding to each active leaf of T_H. Note that there will be a negation node corresponding to an empty active leaf.

Let J be a negation node, corresponding to the active leaf $\leftarrow Q$. Let $Q = \{\neg q_1, \ldots, \neg q_n\}$ where $n \geq 0$. J has n children, one corresponding to each q_i.[1] For $i = 1, \ldots, n$, if q_i is ground then the child corresponding to q_i is the tree node $T_{\leftarrow q_i}$; otherwise the corresponding child is a non-ground node. Non-ground nodes have no children.

Every node has associated with it a *status* (either successful, failed, floundered or indeterminate) according to the following rules. Successful and failed nodes also have an associated *level*.

1. Every non-ground node is *floundered*.

2. (a) If some child of a negation node J is a successful tree node, then we say that J is *failed*. The level of J is the minimum level of all its successful children.

   (b) If every child of a negation node J is a failed tree node, or if J has no children, then we say J is *successful*. The level of J is the least ordinal upper bound of the levels of the children of J. (If J has no children, then it has level 0.)

   (c) If at least one child of a negation node J is a floundered node, and all children of J are not successful, then we say that J is *floundered*.

3. (a) If every child of a tree node T is a failed negation node, or if T is a leaf of Γ_G (i.e. T has no active leaves) then we say T is *failed*. The level of T is $\alpha + 1$, where α is the least ordinal upper bound of the levels of the children of T. (T has level 1 if it has no children.)

   (b) If some child of a tree node T is a successful negation node, then we say T is *successful*. An internal tree node T has level one more than the minimum level of all its successful children. The root tree node may have several associated levels, one for each successful child; the level of the root tree node with respect to such a successful child is one more than the level of the child.

   (c) If at least one child of a tree node T is a floundered negation node, then we say that T is *floundered*.

[1] A negation node corresponding to an empty active leaf will have no children.

4. Any node that can be proved successful, failed or floundered according to the above rules is said to be *well determined.* Any node that is not well determined is said to be *indeterminate.* □

See Figure 4.4 in Example 4.3.1 for an example of a global tree. Note that the definition of the global tree itself is top-down, but that the status of the nodes as successful, failed or floundered is defined bottom-up. The correspondence between the level of a goal and the stage a literal is put in to the well founded model by iterating V_P will be discussed in Section 4.4.

Let L be an active leaf of a tree node in Γ_G. We may say that L is successful, failed, floundered or indeterminate if the corresponding negation node is respectively successful, failed, floundered or indeterminate. We may also say that the goal G is successful, failed, floundered or indeterminate if T_G is respectively successful, failed, floundered or indeterminate in Γ_G.

Definition 4.3.4: Let G be a goal. A *successful branch* of T_G is a branch of T_G that ends at a successful leaf. An *answer substitution* for G is given by $\theta = \theta_1\theta_2\ldots\theta_n$ where the θ_i are the most general unifiers used at each step along a successful branch of T_G. In other words, an answer substitution is the computed most general unifier at a successful leaf. □

A tree node may be both successful and floundered, although no other pair of statuses is possible for a single node. The reason the root tree node is treated differently in rule 3(b) above is that there may be different answer substitutions for the goal that succeed at different levels. Internal tree nodes have ground goals if they are not floundered, and so there cannot be multiple answer substitutions in this case. If we need to distinguish between levels of a goal G with answer substitutions $\theta_1, \theta_2, \ldots$ (each corresponding to a distinct leaf) then we will refer to the level of G *with respect to* θ_i for each i. Failed goals have a unique level.

We may consider a non-ground node as a special type of tree node, in which case Γ_G is a bipartite graph. For every active leaf of a tree node T there will be an edge from T to a negation node. A branch from a negation node to a tree node denotes (negated) membership in the corresponding subgoal.

Definition 4.3.5: Global SLS-resolution is the top-down process of finding all answer substitutions for a goal G using a preferential computation rule. When a derived goal G' containing only negative literals is encountered, the appropriate SLP-trees for the complements of the ground literals in G' are recursively constructed in parallel. □

Global SLS-resolution is, in essence, an appropriate traversal of the global tree for a goal. This process incorporates the traversal of the SLP-trees corresponding to the tree-nodes of the global tree. Selecting a positive literal from a goal corresponds to moving one node deeper in the SLP-tree for the current goal; selecting negative literals corresponds to moving one level deeper in the global tree, by passing through a negation node.

Observe that a goal can have an infinite level even if it involves only finite recursion through negation.

Example 4.3.1: (Van Gelder) Let P be the program

$$\begin{array}{c}
e(s(0), s(s(0))) \\
e(s(X), s(s(X))) \leftarrow e(X, s(X)) \\
e(s(0), 0) \\
e(s(X), 0) \leftarrow e(X, 0)
\end{array}$$

$$\begin{array}{l}
w(X) \leftarrow \neg u(X) \\
u(X) \leftarrow e(Y, X), \neg w(Y)
\end{array}$$

In this example, the integer i is represented by the term $s^i(0)$. The transitive closure of the e predicate defines an ordering among the integers; when (X, Y) is in the transitive closure of e we will write $X < Y$. The ordering is non-standard, with 0 representing the ordinal ω :

$$s^1(0) < s^2(0) < s^3(0) < \cdots < 0.$$

The predicate w is true of an integer j if and only if there are no infinite decreasing sequences starting with j. The predicate u is true precisely when w is false.

Let i, w_i and u_i be abbreviations for $s^i(0)$, $T_{\leftarrow w(s^i(0))}$ and $T_{\leftarrow u(s^i(0))}$ respectively. Then the appropriate SLP-trees and global tree for the goal $\leftarrow w(0)$ are given in Figures 4.1 to 4.4. We use the symbol $\bullet$ to denote negation nodes, and omit the "$\leftarrow$" symbol from goals for clarity.

$$\begin{array}{c}
w(i) \\
| \\
\neg u(i)
\end{array}$$

Figure 4.1: SLP-trees w_i, for $i \geq 0$

$$\begin{array}{c}
u(1) \\
| \\
e(Y, 1), \neg w(Y)
\end{array}
\qquad
\begin{array}{ccc}
 & u(i) & \\
 & | & \\
 & e(Y, i), \neg w(Y) & \\
 & | & \\
 & e(Y', i-1), \neg w(Y'+1) & \\
 & | & \\
 & \vdots & \\
 & | & \\
 & e(Y^{(i-2)}, 2), \neg w(Y^{(i-2)} + (i-2)) & \\
 / & & \backslash \\
\neg w(i-1) & & e(Y^{(i-1)}, 1), \neg w(Y^{(i-1)} + (i-1))
\end{array}$$

Figure 4.2: SLP-trees u_1 and u_i for $i \geq 2$

For $n \geq 1$, the goal $\leftarrow w(s^n(0))$ has level $2n$, and so the goal $\leftarrow w(0)$ has level $\omega + 2$. Note that $\leftarrow w(0)$ has infinite level despite the fact that every branch of the global tree for $\leftarrow w(0)$ is finite. Note that this program does have a well-founded total model, in which $w(0)$ is true, even though it is not locally stratified. $\square$

$$
\begin{array}{ccccccc}
 & u(0) & & & & & \\
 & | & & & & & \\
 & e(Y,0), \neg w(Y) & & & & & \\
/ & & \backslash & & & & \\
\neg w(1) & & e(Y',0), \neg w(Y'+1) & & & & \\
 & / & & \backslash & & & \\
 & \neg w(2) & & e(Y'',0), \neg w(Y''+2) & & & \\
 & & / & & \ddots & & \\
 & & \neg w(3) & & & &
\end{array}
$$

Figure 4.3: SLP-tree u_0

In order for global SLS-resolution to find all answer substitutions, and not get "lost" down an infinite branch of an SLP-tree, an appropriate method for searching SLP-trees (such as breadth first search) is needed. For well-determined goals, global SLS-resolution will (given infinite time) traverse the appropriate global tree. For goals that are indeterminate, global SLS-resolution will recurse through an infinite number of negation nodes. For a discussion of the non-effectiveness of global SLS-resolution see Section 4.7.

There is a close relationship between global SLS-resolution and SLS-resolution. The first difference is that we insist the computation rule be preferential. This restriction is necessary to achieve completeness over the broader class of all programs. (Recall that SLS-resolution is not well-defined for programs that are not locally stratified.)

The second difference is that the definition of SLS-resolution requires a level mapping to be associated with the literals and goals simply in order to define the SLS-tree for a goal. Our construction relaxes this requirement by allowing all subsidiary SLP-trees to be constructed recursively.

Recently, Przymusinski has independently defined a similar extension of SLS-resolution using induction on what he terms the "dynamic stratification" of a program [83]. Dynamic stratification corresponds roughly to what we have called the level of the global tree for a goal. One advantage of our construction is that the level is a *consequence* of the definition of global trees, rather than a *precondition* of its definition.

Another advantage is the explicit representation of the global tree, in which infinite (unsuccessful) branches correspond to indeterminate derivations. Przymusinski's definition only allows selection of subgoals that are known to be well-determined at lower level. However, in a top-down system, the status of subgoals is unknown until they are themselves expanded, and so such a restriction on the selection of subgoals is unlikely to be useful.

Observe that if R is not positivistic, then we will not be able to achieve completeness.

Example 4.3.2: [81] Let P be the program

$$
\begin{aligned}
&p \leftarrow \neg s, q, \neg r \\
&q \leftarrow r, \neg p \\
&r \leftarrow p, \neg q \\
&s \leftarrow \neg p, \neg q, \neg r
\end{aligned}
$$

The well-founded model for P is $\{s, \neg p, \neg q, \neg r\}$. However, consistently expanding the leftmost literal first (i.e. $\neg s$ before q in the first rule) will give us an apparently indeterminate result rather than a successful result for the goal $\leftarrow s$. $\Box$

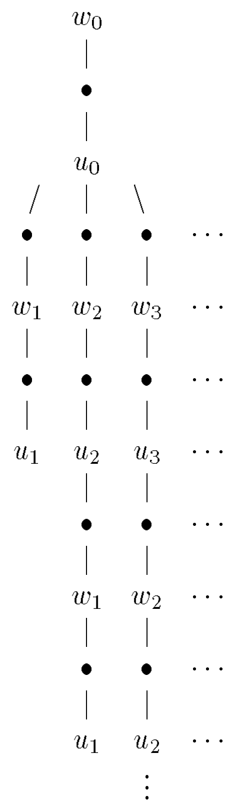

Figure 4.4: Global tree for $\leftarrow w(0)$. Each w_i is successful, and each u_i is failed.

Sequential expansion of ground negative subgoals rather than a parallel expansion is not sufficient to achieve completeness.

Example 4.3.3: Let P be the program

$$\begin{array}{l} p(X) \leftarrow \neg p(f(X)) \\ q \leftarrow \neg p(a), \neg s \\ s \end{array}$$

If a computation rule R chooses the leftmost negative literal first, then the goal $\leftarrow q$ will appear to be indeterminate, although $\neg q$ is in the well-founded (partial) model $\{s, \neg q\}$ for P. However, expanding both negative subgoals in the rule for q in parallel causes q to fail. □

Since only ground negative literals are expanded, they may be processed independently. We could apply the same parallelization to positive literals, but they may generate competing bindings, and resolving such conflicts is beyond the scope of this thesis.

4.4 Ground SLP-Trees and Ground Global Trees

In the following sections we will present our results on the soundness and completeness of global SLS-resolution. First, though, it will be convenient to consider a simplified version of SLP-trees, which we call *ground* SLP-trees. Ground SLP-trees are SLP-trees in which all goals are ground, and rules used in the construction of branches of the SLP-tree are instantiated rules. *Ground* global trees are constructed from ground SLP-trees in a similar way to the construction of global trees from SLP-trees.

We will obtain a new characterization of the well-founded model in terms of ground global trees, that facilitates the proof of soundness and completeness in the general case.

Definition 4.4.1: (Ground SLP-trees) Let G be the ground goal $\leftarrow Q$ and let R be a positivistic computation rule. We define the ground SLP-tree T_G^g for G. The root node of T_G^g is G. If the (ground) goal $H = \leftarrow Q'$ is any node of T_G^g, then its children are obtained as follows.

- If Q' contains a positive literal, then the literal L selected by R from Q' must be positive. In this case, the children of H are all goals K that can be obtained by resolving H with an instantiation of one of the program clauses over the literal L. If there is no such K then H has no children, and is a *dead* leaf.
- If Q' is empty, or contains only negative subgoals then Q' is an *active* leaf.

The *depth* of a node in a ground SLP-tree T is the number of edges in the shortest path from the node to the root of T. $\Box$

There is a structural similarity between SLP-trees and ground SLP-trees that will be made precise in Section 4.5. Since the Herbrand universe of the program may be infinite, there may be infinitely many instantiated rules with a given head. Hence, a ground SLP-tree may have an infinite branching factor at any node. SLP-trees have a finite branching factor.

The *ground global tree* Γ_G^g for a (ground) goal G is defined like the global tree for G except that tree nodes are ground SLP-trees rather than SLP-trees. A goal may be *ground successful*, *ground failed* or *ground indeterminate* in the same way that goals are successful, failed or indeterminate respectively. The difference is that the ground global tree is traversed rather than the global tree. Since all goals appearing in ground SLP-trees are themselves ground, there are no floundered nodes in a ground global tree.

For both global trees and ground global trees, the position of a tree node in the tree does not affect its status. More precisely, if G is a goal, and T_G appears in Γ_H for some goal H, then T_G is successful, failed, indeterminate or floundered in Γ_H if and only if it is respectively successful, failed, indeterminate or floundered in Γ_G, since the status of a tree node depends only on its descendants. A similar property holds for ground global trees. Further, since the level of a goal also depends only on its descendants, the level of G is identical in Γ_H and Γ_G. Hence we shall not refer to the appropriate global tree when discussing the status or level of a goal appearing therein.

On every branch to an active leaf of a ground SLP-tree, every positive literal must eventually be selected, as active leaves contain only negative literals. Further, since there is no interaction between positive literals in a ground goal, the order of selection of the positive literals in a path to an active leaf is not important. Hence, the set of active leaves in T_G^g for a given ground goal G is independent of the computation rule used, as long as the rule is positivistic. Since the status of goals in Γ_G^g depends only on the active leaves of its tree nodes, we may conclude that the semantics induced by Γ_G^g is independent of the computation rule used.[2]

[2] In the more general case of SLP-trees, the set of leaves is again independent of the computation rule used. Such a result may be proved using a "switching lemma." See [60] for details.

In light of this observation, we get Lemma 4.4.1 below. Let $G_1 \uplus G_2$ denote the goal formed as the left-to-right conjunction of the literals in G_1 and G_2. Clearly, $\uplus$ is associative.

Lemma 4.4.1: Let G be the ground goal $\leftarrow p_1, \ldots, p_n, \neg q_1, \ldots, \neg q_m$. Then L is an active leaf of T^g_G if and only if there exist active leaves $L_1, \ldots, L_n$ of $T^g_{\leftarrow p_1}, \ldots, T^g_{\leftarrow p_n}$ respectively such that $L = L_1 \uplus \ldots \uplus L_n \uplus \leftarrow \neg q_1, \ldots, \neg q_m$. Further, the depth of L in T^g_G is equal to the sum over all i of the depths of L_i in $T^g_{\leftarrow p_i}$.

Proof: Since L is an active leaf independent of the computation rule, we may choose any positivistic computation rule for T^g_G. Let $R_1, \ldots, R_n$ be arbitrary positivistic computation rules for $T^g_{\leftarrow p_1}, \ldots, T^g_{\leftarrow p_n}$ respectively. Let R be the computation rule that first simulates R_1, expanding all "descendants" of p_1, then R_2, R_3 and so on until R_n.

L is an active leaf, at finite depth, if and only if each segment of the path to L in T^g_G (corresponding to $R_1, \ldots, R_n$ respectively) is finite, and does not terminate in a dead leaf. This occurs precisely when each $T^g_{\leftarrow p_i}$ has an active leaf, say L_i, and when $L = L_1 \uplus \ldots \uplus L_n \uplus \leftarrow \neg q_1, \ldots, \neg q_m$. The relation between the depths of the leaves is obvious from the construction. ∎

Lemma 4.4.2: Let p be a ground atom, and let G be the goal $\leftarrow p$. Let S^+ be a set of positive ground literals, and S^- a set of negative ground literals. Then

1. $p \in \bigcup_{i=1}^{\infty} \bar{\mathbf{T}}^i_P(S^-) \iff T^g_G$ has an active leaf whose members are all in S^-.
2. $p \in \mathbf{U}_P(S^+) \iff$ every active leaf of T^g_G has a member in $\neg \cdot S^+$.

Proof:

$(1, \Rightarrow)$ If $p \in \bigcup_{i=1}^{\infty} \bar{\mathbf{T}}^i_P(S^-)$ then $p \in \bar{\mathbf{T}}^k_P(S^-)$ for some finite $k \geq 1$. We prove by induction on k that for all p,

$$p \in \bar{\mathbf{T}}^k_P(S^-) \Rightarrow T^g_G \text{ has an active leaf whose members are all in } S^-.$$

Base Case: $p \in \bar{\mathbf{T}}_P(S^-) \iff$ there is some instantiated rule r with head p and (negative) literals from S^- in the body. By definition, this holds if and only if T^g_G has an active leaf (at depth 1) whose members are all in S^-.

Induction Step: Suppose the statement above is true for $k = N$. We show it is also true for $k = N + 1$. Let $p \in \bar{\mathbf{T}}^{N+1}_P(S^-)$. If $p \in \bar{\mathbf{T}}^N_P(S^-)$ then the result follows by the induction hypothesis. Otherwise, there is some instantiated rule r with head p such that all literals $l_1, \ldots, l_n$ in the body of r are in $\bar{\mathbf{T}}^N_P(S^-)$. Since the only negative elements of $\bar{\mathbf{T}}^N_P(S^-)$ are actually in S^-, all of the negative literals in $\{l_1, \ldots, l_n\}$ are in S^-. By the induction hypothesis, for each of the positive literals l_i, (for $i = 1, \ldots, m$ with $m \leq n$), $T^g_{\leftarrow l_i}$ has an active leaf (say L_i) whose members are all in S^-. Let $L = L_1 \uplus \ldots \uplus L_m \uplus \leftarrow l_{m+1}, \ldots, l_n$. Then L must be an active leaf of T^g_G by Lemma 4.4.1. Further, $L \subseteq S^-$, thus demonstrating the result for $k = N + 1$.

$(1, \Leftarrow)$ This argument is by induction on the depth of the active leaf. We show that if T^g_G has an active leaf at depth d, all of whose members are in S^-, then $p \in \bar{\mathbf{T}}^d_P(S^-)$. The base case of this induction is identical to the base case of the previous argument.

Induction Step: Suppose the statement is true for all $d \leq N$. We show it is true for $d = N+1$. If T^g_G has an active leaf L at depth $N+1$ such that $L \subseteq S^-$, then some child G' of G has L as an active leaf at depth N. Let r be the instantiated rule used in deriving G' from G, given by $p \leftarrow p_1, \ldots, p_n, \neg q_1, \ldots, \neg q_m$, so that $G' = \{p_1, \ldots, p_n, \neg q_1, \ldots, \neg q_m\}$. By Lemma 4.4.1, for every positive literal p_i in G', $T^g_{\leftarrow p_i}$ has an active leaf at depth at most N, whose members are all in L, and hence in S^-. By our induction hypothesis, $p_i \in \bar{\mathbf{T}}^N_P(S^-)$. Each $\neg q_i$ is in S^- (since it is in L), and hence in $\bar{\mathbf{T}}^N_P(S^-)$. Thus $p \in \bar{\mathbf{T}}^{N+1}_P(S^-)$.

(2, $\Rightarrow$) Suppose $p \in \mathbf{U}_P(S^+)$. Then for every instantiated rule r with head p, either some positive literal in the body is also in $\mathbf{U}_P(S^+)$, or some negative literal is in $\neg \cdot S^+$. Hence every goal appearing in T^g_G has either a positive literal in $\mathbf{U}_P(S^+)$ or a negative literal in $\neg \cdot S^+$. Since active leaves have no positive literals, it follows that every active leaf of T^g_G contains a negative literal in $\neg \cdot S^+$.

(2, $\Leftarrow$) Suppose every active leaf in T^g_G has a member in $\neg \cdot S^+$. We show $p \in \mathbf{U}_P(S^+)$ by constructing an unfounded set U (with respect to S^+) containing p. Since $\mathbf{U}_P(S^+)$ contains all unfounded sets, the result will follow. Let U be the set of ground atoms defined by

$$U = \{q : \text{Every active leaf of } T^g_{\leftarrow q} \text{ has a member in } \neg \cdot S^+\}$$

$p \in U$ by assumption. Let q' be an arbitrary element of U. We claim that every goal appearing in $T^g_{\leftarrow q'}$ contains either

- a positive literal p' such that $p' \in U$, or
- a negative literal l such that $l \in \neg \cdot S^+$.

We prove the claim by contradiction. Suppose the contrary, i.e., that for some goal $H = \leftarrow p_1, \ldots, p_n, \neg q_1, \ldots, \neg q_m$ in $T^g_{\leftarrow q'}$, no p_i is in U and no $\neg q_j$ is in $\neg \cdot S^+$. Then for each p_i there must be an active leaf (say L_i) of $T^g_{\leftarrow p_i}$ containing no members of $\neg \cdot S^+$ (by the definition of U, and by Lemma 4.4.1). Hence there must be an active leaf L of $T^g_{\leftarrow q'}$ that is a descendent of H, and is given by $L_1 \uplus \ldots \uplus L_n \uplus \leftarrow \neg q_1, \ldots, \neg q_m$. But no element of L is in $\neg \cdot S^+$, contradicting the assumption that $q' \in U$, and thus proving the claim.

In particular, every child of $\leftarrow q'$ in $T^g_{\leftarrow q'}$ satisfies the above claim. But the children of $\leftarrow q'$ are simply the bodies of all instantated rules with q' as head. Any literal that makes the claim above true for such a rule body is a witness of unusability for that rule; the claim shows that every rule for q' has a witness of unusability. Since q' is arbitrary, by the definition of unfounded sets, U is unfounded with respect to S^+ and the result follows.

∎

The above results are false if we do not insist that S^+ and S^- contain only positive and negative literals respectively. However, as the following result shows, Lemma 4.4.2 is sufficient for our purposes.

Lemma 4.4.3: Let l be a ground literal in $M_{WF}(P)$. Suppose $l \in I_{\alpha+1}$ according to the iteration of Definition 4.2.3. Then

- If l is positive, then $l \in \bigcup_{i=1}^{\infty} \bar{\mathbf{T}}_P^i(\mathbf{Neg}(I_\alpha))$.
- If l is negative, then $l \in \neg \cdot \mathbf{U}_P(\mathbf{Pos}(I_\alpha))$.

Proof: The proof is by transfinite induction on α.

Case 1: α is a successor ordinal. Let $\alpha = \beta + 1$ and assume the result for β.

For the first part, we know $l \in \bigcup_{i=1}^{\infty} \bar{\mathbf{T}}_P^i(I_\alpha)$ and so $l \in \bigcup_{i=1}^{\infty} \bar{\mathbf{T}}_P^i(I'_\alpha)$ for some finite subset I'_α of I_α. (This compactness property follows from the finiteness of the bodies of instantiated rules.) By hypothesis, and since I'_α is finite, there is some $k \geq 0$ such that all the positive literals in I'_α are in $\bar{\mathbf{T}}_P^k(\mathbf{Neg}(I_\beta))$. ($\bar{\mathbf{T}}(S) \supseteq S$ implies that the finite union to the k^{th} term simplifies to the final term.) By monotonicity $\mathbf{Pos}(I'_\alpha) \subseteq \bar{\mathbf{T}}_P^k(\mathbf{Neg}(I_\alpha))$. Hence $l \in \bigcup_{i=1}^{\infty} \bar{\mathbf{T}}_P^i(\bar{\mathbf{T}}_P^k(\mathbf{Neg}(I_\alpha)))$, and the result follows.

For the second part, suppose $l = \neg p$. We know that $\mathbf{Neg}(I_\alpha) \subseteq \neg \cdot \mathbf{U}_P(\mathbf{Pos}(I_\beta))$ by our induction hypothesis. Hence, by monotonicity, $\mathbf{Neg}(I_\alpha) \subseteq \neg \cdot \mathbf{U}_P(\mathbf{Pos}(I_\alpha))$. Let w be any witness of unusability for a rule r having head p, with respect to I_α. If $w \in \mathbf{Pos}(I_\alpha)$ then it certainly remains a witness with respect to $\mathbf{Pos}(I_\alpha)$. If $w \in \mathbf{Neg}(I_\alpha)$ then $w \in \neg \cdot \mathbf{U}_P(\mathbf{Pos}(I_\alpha))$ by the above, and hence w is still a witness for r with respect to $\mathbf{Pos}(I_\alpha)$. Hence $p \in \mathbf{U}_P(\mathbf{Pos}(I_\alpha))$ and $l \in \neg \cdot \mathbf{U}_P(\mathbf{Pos}(I_\alpha))$.

Case 2: α is a limit ordinal. The proof in this case is a simple extension of the arguments in the successor ordinal case. The case $\alpha = 0$ is trivial since $I_\alpha = \emptyset$. For the first part, where l is positive, the proof is essentially the same once we observe that I'_α, being finite, must be a subset of I_γ for some successor ordinal $\gamma < \alpha$.

For the second part, observe that if $l \in \mathbf{Neg}(I_\alpha)$ then $l \in \mathbf{Neg}(I_{\beta+1})$ for some successor ordinal $\beta + 1$. By hypothesis, $l \in \neg \cdot \mathbf{U}_P(\mathbf{Pos}(I_\beta))$, and since l is arbitrary, it follows that $\mathbf{Neg}(I_\alpha) \subseteq \bigcup_{\beta<\alpha} \neg \cdot \mathbf{U}_P(\mathbf{Pos}(I_\beta))$. We can extend the argument of the successor ordinal case to show that $\mathbf{Neg}(I_\alpha) \subseteq \bigcup_{\beta<\alpha} \neg \cdot \mathbf{U}_P(\mathbf{Pos}(I_\beta))$ implies

$$\mathbf{Neg}(I_\alpha) \subseteq \neg \cdot \mathbf{U}_P(\bigcup_{\beta<\alpha} \mathbf{Pos}(I_\beta)),$$

by monotonicity, and hence $\mathbf{Neg}(I_\alpha) \subseteq \neg \cdot \mathbf{U}_P(\mathbf{Pos}(I_\alpha))$. The remainder of the proof is identical.

∎

Lemma 4.4.4: Let l be a ground literal. Then

- If l is positive, then $l \in I_{\alpha+1} \iff l \in \bigcup_{i=1}^{\infty} \bar{\mathbf{T}}_P^i(\mathbf{Neg}(I_\alpha))$.
- If l is negative, then $l \in I_{\alpha+1} \iff l \in \neg \cdot \mathbf{U}_P(\mathbf{Pos}(I_\alpha))$.

Proof: The implications from left to right follow from Lemma 4.4.3. The implications from right to left follow by monotonicity. ∎

We now demonstrate the precise correspondence between the well-founded semantics and ground global trees.

Theorem 4.4.5: Let p be a ground atom, G the goal $\leftarrow p$ and α a countable ordinal. Then

- G is ground successful at level $\leq \alpha \iff p \in I_\alpha$.
- G is ground failed at level $\leq \alpha \iff \neg p \in I_\alpha$.

Proof: The argument is by induction on α.

Case 1: α is a successor ordinal. Suppose $\alpha = \beta + 1$, and suppose the statement is true for the ordinal β.

G is ground successful at level $\leq \alpha$

$$
\begin{array}{ll}
\iff & T^g_G \text{ has an active leaf } L = \leftarrow \neg p_1, \ldots, \neg p_n \text{, say, such that} \\
 & \text{each } T^g_{\leftarrow p_i}\ (i = 1, \ldots, n) \text{ is ground failed at level} \leq \beta \\
\iff & L \subseteq \mathbf{Neg}(I_\beta)\text{, by hypothesis} \\
\iff & p \in \bigcup_{i=1}^{\infty} \bar{\mathbf{T}}^i_P(\mathbf{Neg}(I_\beta))\text{, by Lemma 4.4.2} \\
\iff & p \in I_\alpha\text{, by Lemma 4.4.4}
\end{array}
$$

G is ground failed at level $\leq \alpha$

$$
\begin{array}{ll}
\iff & \text{every active leaf of } T^g_G \text{ contains a literal } \neg q \text{ such that} \\
 & T^g_{\leftarrow q} \text{ is ground successful at level} \leq \beta \\
\iff & \text{every active leaf of } T^g_G \text{ has a member whose complement is} \\
 & \text{in } \mathbf{Pos}(I_\beta)\text{, by hypothesis} \\
\iff & p \in \mathbf{U}_P(\mathbf{Pos}(I_\beta))\text{, by Lemma 4.4.2} \\
\iff & \neg p \in I_\alpha\text{, by Lemma 4.4.4}
\end{array}
$$

Case 2: α is a limit ordinal. The truth of the above statements for $\alpha = 0$ is trivial. By the construction of global trees, goals can only be successful or failed at a level that is a successor ordinal. Also, the stage of every ground literal must be a successor ordinal, as observed in Definition 4.2.3.

Let us consider first the implication from left to right. For limit ordinals $\alpha > 0$, G is ground successful (respectively, ground failed) at level $\leq \alpha$ implies that G is ground successful (ground failed) at level β for some successor ordinal $\beta < \alpha$. Hence by hypothesis, p ($\neg p$) is in I_β and hence in I_α by monotonicity.

Conversely, p (respectively $\neg p$) is in I_α implies that p ($\neg p$) is in I_β for some successor ordinal $\beta < \alpha$. Hence p is ground successful (ground failed) at level $\leq \beta$ by hypothesis, and the result follows since $\beta < \alpha$.

∎

Corollary 4.4.6: Let p be an atom, G the goal $\leftarrow p$ and α a countable ordinal. Then

- G is ground successful at level $\alpha \iff stage_p = \alpha$
- G is ground failed at level $\alpha \iff stage_{\neg p} = \alpha$

▮

Theorem 4.4.7: Let $Q = \{p_1, \ldots, p_n, \neg q_1, \ldots, \neg q_m\}$ be an arbitrary ground query, and let G be the goal $\leftarrow Q$.

- G is ground successful $\iff M_{WF}(P) \models Q$
- G is ground failed $\iff M_{WF}(P) \models \neg Q$
- G is ground indeterminate $\iff M_{WF}(P) \not\models Q$ and $M_{WF}(P) \not\models \neg Q$

Proof: G is ground successful if and only if each p_i is ground successful, and each q_j is ground failed, by Lemma 4.4.1. This happens precisely when each p_i and each $\neg q_j$ is in $M_{WF}(P)$, by Theorem 4.4.5, i.e., when $M_{WF}(P) \models Q$.

Similarly, G is ground failed if and only if some q_j is ground successful or some p_i is ground failed, again by Lemma 4.4.1. This happens precisely when some $\neg p_i$ or some q_j is in $M_{WF}(P)$, by Theorem 4.4.5, i.e., when $M_{WF}(P) \models \neg Q$. ▮

4.5 Soundness

Now that the correspondence between ground global trees and the well founded semantics has been established, we investigate the correspondence between ground global trees and (general) global trees.

Lemma 4.5.1: Let P be a program and G a goal. Let L' be an arbitrary active leaf of T_G, with computed most general unifier θ. Then for every ground substitution δ for $G\theta$, $T^g_{G\theta\delta}$ has an active leaf L that is an instance of L'.

Proof: Let $r_1, \ldots, r_n$ be the rules used in the branch ending in L' in T_G, and let $\theta_1, \ldots, \theta_n$ be the corresponding most general unifiers. By definition, $\theta = \theta_1\theta_2 \ldots \theta_n$. We prove the result by induction on the depth, n.

When $n = 0$, G is itself the only active leaf of T_G, with the identity computed most general unifier. $G\delta$ is then the only active leaf of $T^g_{G\delta}$, and is clearly an instance of G.

Suppose the result is true for $n = k$. We show it is true for $n = k + 1$. Let G be $\leftarrow p_1, \ldots, p_l, \neg q_1, \ldots, \neg q_m$, and suppose L' is at depth $k + 1$ in T_G. Suppose that r_1 is the rule $p \leftarrow b_1, \ldots, b_{l'}, \neg c_1, \ldots, \neg c_{m'}$, and that p unifies with p_i using most general unifier θ_1. Then the resultant goal G' is

$$\leftarrow (p_1, \ldots, p_{i-1}, p_{i+1}, \ldots, p_l, \neg q_1, \ldots, \neg q_m, b_1, \ldots, b_{l'}, \neg c_1, \ldots, \neg c_{m'})\theta_1.$$

(The order of literals in a goal is unimportant.)

L' is a leaf of $T_{G'}$ at depth k, with computed most general unifier $\theta' = \theta_2 \ldots \theta_{k+1}$. Let γ be a substitution such that $G'\theta'\delta\gamma$ is ground. By our induction hypothesis, there is an active leaf L of $T^g_{G'\theta'\delta\gamma}$ that is an instance of L'. But $G'\theta'\delta\gamma$ is a child of $G\theta\delta$ in $T^g_{G\theta\delta}$ using the rule $r_1\theta\delta\gamma$ which must be ground. Hence $T^g_{G\theta\delta}$ has a leaf L that is an instance of L'. Since δ is arbitrary, the result follows. ▮

We now investigate the converse of Lemma 4.5.1.

Lemma 4.5.2: Let P be a program, G a goal, and δ a ground substitution for G. Let L be an arbitrary active leaf of $T^g_{G\delta}$. Then there exists an active leaf L' of T_G, with computed most general unifier θ, such that L is an instance of L' and θ is more general then δ.
Proof: (Sketch)

We may consider the branch to L in $T^g_{G\delta}$ as an *unrestricted derivation* from G, i.e., a derivation in which we do not insist that unifiers be most general. (In the ground SLP-tree, such unifiers always make the resulting goal ground.) The proof of this lemma is then very similar to the proof of the "mgu lemma" in [60], and the details are omitted here. ∎

Lemma 4.5.3: Let G be a goal. Then

- If G is successful with answer substitution θ, then $G\theta\delta$ is ground successful for every ground substitution δ for $G\theta$.
- If G is failed then $G\delta$ is ground failed for every ground substitution δ for G.

Proof: The proof is by induction on the level of G with respect to θ. Since no goals succeed or fail at limit ordinals (including the base case, 0), the result for limit ordinals is trivial. We now consider the successor ordinal case.

(First part) Suppose that G is successful with respect to θ at level α, and that $L = \leftarrow \neg p_1, \ldots, \neg p_m$ is the successful leaf with answer substitution θ. Then each p_i must be ground, and each $\leftarrow p_i$ failed at level $\beta_i < \alpha$. By our induction hypothesis, each $\leftarrow p_i$ is ground failed.

Since L is ground, L must appear as a leaf of $T^g_{G\theta\delta}$ for every ground substitution δ, by Lemma 4.5.1. Finally, since each $\leftarrow p_i$ is ground failed, $G\theta\delta$ is ground successful.

(Second part) Suppose G is failed at level α, and that $L' = \leftarrow \neg p_1, \ldots, \neg p_m$ is an arbitrary active leaf of T_G. Then some p_i must be ground, and $\leftarrow p_i$ successful at level $\beta_i < \alpha$. By our induction hypothesis, $\leftarrow p_i$ is ground successful. Since L' is arbitrary, every active leaf of T_G contains such a ground subgoal.

Let δ be an arbitrary ground substitution for G. Every active leaf of $T^g_{G\delta}$ is an instance of an active leaf in T_G, by Lemma 4.5.2. Hence every active leaf of $T^g_{G\delta}$ contains a literal $\neg q$ such that $\leftarrow q$ is ground successful. By definition, $G\delta$ is ground failed. Since δ is arbitrary, the result follows. ∎

We may now prove the soundness of global SLS-resolution.

Theorem 4.5.4: (Soundness of global SLS-resolution) Let P be a program, let Q be a query, and let G be the goal $\leftarrow Q$. Then

1. If G is successful with answer substitution θ then $M_{WF}(P) \models \forall(Q\theta)$.
2. If G is failed then $M_{WF}(P) \models \forall(\neg Q)$.

Proof: (First part) Let δ be a ground substitution for $G\theta$. By Lemma 4.5.3, G is successful with answer substitution θ implies that $G\theta\delta$ is ground successful. By Theorem 4.4.7, $M_{WF}(P) \models Q\theta\delta$. Since δ is arbitrary, it follows that $M_{WF}(P) \models \forall(Q\theta)$.

(Second part) By Lemma 4.5.3, G is failed implies that $G\delta$ is ground failed for all ground substitutions δ for G. By Theorem 4.4.7, $M_{WF}(P) \models \neg Q\delta$ for all such δ, and hence $M_{WF}(P) \models \forall(\neg Q)$. ∎

4.6 Completeness

In this section we address the completeness of global SLS-resolution. First, though, we remark that our restriction to non-floundering goals in this section does not result in any loss of expressive power. To see this point, assume we have a unary predicate *term* that binds its argument to every term in the Herbrand universe of the program. By adding the atom $term(X)$ to the body of each rule for all variables X in the rule, and by adding this atom also to the goal, one can guarantee that floundering will not occur, without changing the well-founded model of the program. Incidentally, this observation shows that even when restricted to non-floundering literals, the well-founded model is not, in general, recursively enumerable.

Constructing such a predicate *term* is simple: For each constant c appearing in the program include the rule $term(c)$, and for each n-ary function symbol f in the program include the rule

$$term(f(X_1, \ldots, X_n)) \leftarrow term(X_1), \ldots, term(X_n).$$

Lemma 4.6.1: (Lifting lemma) Let P be a program, G a non-floundering goal and let δ range over all ground substitutions for G. Then

- If for some δ, $G\delta$ is ground successful then G is successful with an answer substitution θ more general than δ.
- If for all δ, $G\delta$ is ground failed, then G is failed.

Proof: The proof is by induction on the level of $G\delta$. (For the second part, the induction is on the *maximum* over all δ of the level of $G\delta$.) Again, since goals cannot succeed at limit ordinals, the limit ordinal case, including the base case 0, is trivial.

(First part) Suppose $G\delta$ is ground successful at level α, and let $L = \leftarrow \neg p_1, \ldots, \neg p_n$ be a successful leaf in $T_{G\delta}$. By Lemma 4.5.2, T_G has a leaf $L' = \leftarrow \neg p'_1, \ldots, \neg p'_n$ such that each p_i is an instance of p'_i, and the computed most general unifier θ at L' is more general than δ.

Since $G\delta$ is ground successful, each $\leftarrow p_i$ is ground failed at strictly lower level. If $p'_i = p_i$ (i.e., p'_i is ground) then p'_i is failed by our induction hypothesis. If p'_j was not ground for some j, then the negation node corresponding to L would have a non-ground child. Since no $\leftarrow p'_i$ can be successful (it must be either failed or floundered), this would contradict our assumption that G is not floundered.

Hence L' is ground and every $\leftarrow p'_i$ is failed. Thus, G is successful with answer substitution θ more general than δ.

(Second part) Let $L' = \leftarrow \neg p'_1, \ldots, \neg p'_n$ be an arbitrary leaf of T_G, with computed most general unifier θ. Let γ be a ground substitution for $G\theta$, and let $\delta = \theta\gamma$. Then $T^g_{G\delta}$ has an active leaf $L = \leftarrow \neg p_1, \ldots, \neg p_n$ that is an instance of L', by Lemma 4.5.1. Since $G\delta$ is ground failed by assumption, some $\leftarrow p_i$ is ground and successful at strictly lower level.

Consider $\leftarrow p'_i$, which is more general than $\leftarrow p_i$. If p'_i is ground, then $p'_i = p_i$ and so $\leftarrow p'_i$ is successful by the induction hypothesis. If p'_i is not ground, then since G is not floundered, some other p'_j is ground and successful. In either case, L' is a failed leaf in T_G. Since our choice of L' was arbitrary, all active leaves of T_G are failed, and hence G is failed. ∎

The restriction to non-floundering goals in the above lemma cannot be omitted due to programs of the form

$$p(X) \leftarrow \neg q(f(X))$$
$$q(a)$$

for which the goal $\leftarrow p(X)$ flounders, while every ground instance of this goal succeeds. This example indicates that disallowing floundering goals altogether is perhaps too harsh. However, if we add the rule $q(f(a))$ to the program above then we are faced with somehow trying to represent the success set for $\leftarrow p(X)$ as "X may be anything except $f(a)$," a concept that requires a broader notion of answer substitution. Some results in this direction have been presented in [58], and some more recent work has described a process called "constructive negation" in which negative subgoals are used to generate negative bindings [21, 84]. Whether such methods will be useful in practice, or whether ground negation is sufficient for most purposes remains to be seen. Restricting programs and goals to be range-restricted (Definition 2.1.8), for example, guarantees freedom from floundering.

Before we present our completeness result, we first note that we shall prove completeness with respect to the augmented program (Section 3.13). Certain anomalies occur when working only with Herbrand interpretations.

Example 4.6.1: Consider the program P given by the single rule

$$p(a)$$

The only Herbrand model of P is $\{p(a)\}$ and so $\forall X\ p(X)$ is true in all Herbrand models, although it is not a logical consequence of the program. However, if we add the apparently unrelated fact $q(b)$ to P, $\forall X\ p(X)$ becomes false in some Herbrand models. Further, no resolution-type procedure will give the identity answer substitution for the query $\{p(X)\}$, and so we would have little hope of achieving completeness with respect to the query $\{p(X)\}$ on this program. $\forall X\ p(X)$ is not true in all Herbrand models of the augmented program P', and so use of the augmented program overcomes such anomalous behaviour. $\square$

Theorem 4.6.2: (Completeness of global SLS-resolution) Let P be a program, and let $G = \leftarrow Q$ be a non-floundering goal involving only symbols from P. Let P' be the augmented version of P. Let ϕ be a substitution for the variables of Q. Then

1. If $M_{WF}(P) \models \exists Q$ then G succeeds
2. If $M_{WF}(P) \models \forall(\neg Q)$ then G is failed
3. If $M_{WF}(P') \models \forall(Q\phi)$ then G succeeds with an answer substitution more general than ϕ.

Proof:

1. If $M_{WF}(P) \models \exists Q$, then $M_{WF}(P) \models Q\delta$ for some ground substitution δ. By Theorem 4.4.7, $G\delta$ is ground successful. By Lemma 4.6.1 G is succeessful.
2. If $M_{WF}(P) \models \forall\neg Q$ then for every ground substitution δ, $M_{WF}(P) \models \neg Q\delta$. Hence, by Theorem 4.4.7 $G\delta$ is ground failed. By Lemma 4.6.1 G is failed.
3. (We assume ϕ does not mention the symbols $\bar{f}$ or $\bar{c}$. Extending the proof below in the case that $\bar{f}$ or $\bar{c}$ does appear in ϕ is straightforward.)

   Let $\{x_0, \ldots, x_n\}$ be the variables appearing in $Q\phi$. Let δ be the (ground) substitution $\{x_0|\bar{c}, x_1|\bar{f}(\bar{c}), \ldots x_n|\bar{f}^n(\bar{c})\}$ of terms from the Herbrand universe of the augmented program for the variables in $Q\theta$. Then $M_{WF}(P') \models Q\phi\delta$, and so $G\phi\delta$ is ground successful by Theorem 4.4.7.

By Lemma 4.6.1, G succeeds with an answer substitution θ such that for some substitution γ

$$\theta\gamma = \phi\delta.$$

Now θ cannot contain any substitutions involving $\bar{f}$ or $\bar{c}$ since G contains only symbols from P, and the predicate $\bar{p}$ in P' appears nowhere in P. Hence the only occurrences of $\bar{f}$ and $\bar{c}$ in the left side of the equality above are in γ. Let γ' be formed from γ by replacing every occurrence of $\bar{f}^i(\bar{c})$ by the variable x_i. Then $\theta\gamma' = \phi$, and so θ is indeed more general than ϕ.

▮

We cannot substitute P for P' in the third item in Theorem 4.6.2 as illustrated by Example 4.6.1. Some texts (for example [60]) make the implicit assumption that extra ground terms exist in order to prove completeness results. The purpose of the augmented program is to formally include sufficiently many such ground terms in the Herbrand universe.

Corollary 4.6.3: For non-floundering goals $G = \leftarrow Q$,

$$T_G \text{ is indeterminate} \Leftrightarrow \text{neither } M_{WF}(P) \models \exists(Q) \text{ nor } M_{WF}(P) \models \forall(\neg Q). \quad ▮$$

4.7 Discussion

The well-founded semantics is a declarative semantics that unifies a number of approaches in a robust fashion. In order to be able to *use* well-founded negation in logic programs, a corresponding procedural semantics is necessary. This chapter presents such a procedural semantics.

Global SLS-resolution is not effective, as discussed below. However, for certain subclasses of programs, global SLS-resolution is effective. For acyclic programs [5, 20] or, more generally, for modularly acyclic programs [97] global SLS-resolution is effective (see Chapter 7). In the absence of function symbols, the Herbrand Base is finite, and so effective procedures exist.[3] For modularly stratified function-free programs global SLS-resolution may be made effective by using the memoing techniques of [51, 106] to prune loops (see Chapter 5).

There are three sources of non-effectiveness in global SLS-resolution:

1. Infinite branches of an SLP-tree are treated as failed.
2. The SLP-tree for a goal may have an infinite number of branches, (although only a finite branching factor at any particular depth).
3. If a goal is indeterminate, global SLS-resolution will recurse infinitely through negation.

Although a preferential computation rule selects positive literals ahead of negative literals, in practice a sub-process may be spawned to expand a negative subgoal as soon as it becomes ground. Such spawning will not only improve performance but may allow the earlier pruning of long branches.

Note that SLDNF-resolution using a safe computation rule is sound with respect to the well-founded semantics for all programs. However, even with a preferential computation rule, SLDNF-resolution is incomplete as it does not treat infinite branches of an SLP-tree as failed.

[3] A polynomial time algorithm for constructing the well-founded model for function-free programs, that is bottom-up in nature, is given in the full version of [118].

Chapter 5

Evaluation: Memoing and Magic Sets

5.1 Background

Much recent work has concerned defining the semantics of negation in deductive databases. The "perfect model semantics" [82] has been generally accepted as natural, and is the basis for several experimental deductive database systems. Unfortunately, the perfect model semantics applies only to programs that are stratified (or locally stratified).

As discussed in Chapter 3, there are interesting programs that are not stratifiable but for which a natural, unambiguous semantics exists.

The well-founded semantics is a three-valued semantics. Literals may be *true*, *false* or *undefined*. Nevertheless, there are many cases where a non-stratified program has a total semantics, i.e., a semantics in which every ground literal is either true or false. Allowing programs that have some literals *undefined* may not be desirable, since handling this extra truth value places an extra burden on the query evaluation procedure. In many cases, two truth values suffice to model the situation under consideration. So we desire a condition on the program, more general than stratification, that ensures that the well-founded semantics is two-valued. Recently Przymusinska and Przymusinski [81] have isolated the class of *weakly stratified* programs as such a class. *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.

However, weakly stratified programs cannot, in general, be evaluated in a subgoal-at-a-time fashion. When evaluated left-to-right, say, a subgoal may go into an infinite loop through negation even though some later subgoal fails. Since subgoal-at-a-time computation is important for efficient evaluation or possible parallelization, we would like to identify a class of weakly stratified programs that can be evaluated one subgoal at a time.

In this paper we propose such a class, which we term the class of *modularly stratified* programs. Every modularly stratified program is weakly stratified, but the converse is false. For weakly stratified programs (and hence for modularly stratified programs) the well-founded semantics is total (i.e., makes every ground literal either true or false). The well-founded semantics and the stable model semantics coincide for weakly stratified programs, a consequence of the fact that the well-founded model is total. Modularly stratified programs also allow subgoal-at-a-time evaluation.

A program is *modularly stratified* if and only if its mutually recursive components are *locally stratified* once all instantiated rules with a false subgoal that is defined in a "lower" component are removed.

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 (considered as inputs) satisfying certain intuitive constraints. This issue is related to the notion of compositionality (see Section 3.8).

Kemp and Topor, and Seki and Itoh have proposed (similar) extensions of the QSQR/SLD top-down query evaluation procedure of Vieille [120, 121] from Horn programs to the class of stratified programs [51, 106]. We further generalize their methods, called QSQR/SLS-resolution in [51], to the class of modularly stratified programs.

In the absence of negation, bottom-up computation may be made at least as efficient as a top-down approach by the use of an optimization technique known as magic sets [112]. Several authors have considered the problem of extending the magic sets method to stratified programs.

The approach of [8, 9] is to relabel predicates in a stratified program in such a way that the magic transformation results in a stratified program. This method does not work for all stratified programs, though. More recently, these authors have considered a "structured" bottom-up method that uses control information in order to sequence rule execution [10].

The approach of [14] is similar in nature to this structured bottom-up method. The authors perform the magic rewriting, which may result in an unstratified program, and then impose constraints on the order of evaluation of rules, in the form of regular expressions. These constraints ensure that a subgoal is "fully evaluated" before a predicate depending negatively on that subgoal is considered.

Kerisit and Pugin also consider extending magic sets to programs with stratified negation [52]. They define a property called weak stratification[1] and demonstrate that the magic rewriting of a stratified program must be weakly stratified. The evaluation of weakly stratified programs is analogous to the evaluation of stratified programs in that one computes an iterated least fixpoint, lowest level first.

In [18], Bry outlines a magic sets method for what he calls constructively consistent programs. Unfortunately, the section on magic sets in that paper is very brief. Only after the paper [96] was written did we become aware of the details of Bry's work [17]. The class of constructively consistent programs also generalizes the class of stratified programs, and constructively consistent programs have a two-valued semantics. Bry's method handles negative dependencies by storing clauses rather than atoms. Rules are unfolded, leaving negative literals in the bodies, and these unfolded rules effectively control the order of rule execution.

Unfolded rules provide a less compact representation than facts. Also, using unfolded rules can lead to repeated computation of the same subgoals on different branches of the computation tree. For example, unfolding the rule

$$q \leftarrow p, p$$

will duplicate the computation for p.

Our approach is similar in nature to these proposals, although it is not, strictly speaking, a generalization of them. Rather than using control information or regular expressions to enforce

[1]This concept is different from Przymusinski's definition of weak stratification, and was proposed independently. Kerisit and Pugin define weak stratification by removing from the conditions on stratified programs the restriction that positive literals in the body have level no higher than the level of the head.

ordering constraints, we incorporate information about negative dependencies into the bottom-up evaluation in the form of a *depends* meta-predicate. This formalism allows us to handle the class of modularly stratified programs. The levels of a stratified program are used at *compile-time* by the previous proposals to control the order of rule firing. By maintaining dependence information within the evaluation at *run-time* we can handle a larger class of programs.

We present a magic sets transformation for modularly stratified programs that enables efficient computation bottom-up. In fact, we demonstrate a one-to-one correspondence between steps of (our extension of) QSQR/SLS-resolution and our magic-sets method. See [113] for a discussion of why bottom-up methods are likely to be preferable to top-down methods.

In this chapter we assume all programs are function-free normal programs. The definition of modular stratification in Section 5.2 does not depend on the assumption of function-freeness, although both our QSQR/SLS procedure of Section 5.3.2 and the magic sets method presented in Section 5.4.3 may not terminate for programs with function symbols. We shall also make the assumption that programs are *range restricted* (Definition 2.1.8).

We place an ordering on the rules in a program, for example the top-to-bottom ordering of the rules as written. Thus we number the rules $r_1, \ldots, r_n$ where there are n rules in total.

We do allow one piece of syntactic sugar in negative subgoals, namely the "don't care" variable. For example, in a family database we may write the rule

$$childless(X) \leftarrow person(X), \neg parent(X, \_)$$

as shorthand for

$$childless(X) \leftarrow person(X), \neg \exists Y\, parent(X, Y)$$

which can easily be rewritten as a normal program.

5.2 Modularly Stratified Programs

Example 5.2.1: Consider the program P consisting of the rule

$$w(X) \leftarrow m(X, Y), \neg w(Y)$$

together with some facts about m. P is a game-playing program [53] in which a position X is "winning" $[w(X)]$ if there is a move from X to a position Y $[m(X, Y)]$ and Y is a losing position $[\neg w(Y)]$. (This is the same program as in Example 3.7.3.)

P is not stratified or even locally stratified. In fact Kolaitis [53] has shown that no stratified function-free program can express the intended semantics of P. The intuition behind this expressiveness result is that stratified programs have a fixed number of strata through which one can recurse through negation. However, we would like to determine whether positions in an *arbitrary* game (given by a move relation m) are winning or losing. In particular to answer queries for a game with game-tree depth d we need to recurse through negation d times. Since d can be arbitrarily large, we cannot use a stratified program to answer this type of query.

This lack of expressive power is the main motivation for generalizing the class of stratified programs. As will be shown later, P is modularly stratified when m is acyclic.[2] □

Definition 5.2.1: We say a predicate p is *depends upon* a predicate q if there is a sequence of rules $t_0, \ldots, t_{n-1}$ with predicates $p_0, \ldots, p_{n-1}$ in the head, respectively, such that

[2] I.e., when the game cannot have repeated positions.

1. $p = p_0$ and $q = p_n$, and
2. for $i = 1, \ldots, n$, p_i appears (positively or negatively) in the body of t_{i-1}.

We say p depends on q through k negations if exactly k of the appearances of $p_1, p_2, \ldots, p_n$ in $t_0, \ldots, t_{n-1}$, respectively, are negative. We say p depends *negatively* on q if p depends on q through at least one negation. A predicate p is *recursive* in a predicate q if p depends upon q and q depends upon p. $\Box$

Definition 5.2.2: Let F be a component (i.e., a subset of the rules) of a program P. We say F is a *complete* component if for every predicate p appearing in the head of a rule in F,

- all rules in P with head predicate p are in F, and
- if p is recursive in a predicate q, then all rules in P with head predicate q are in F.

If the predicate p appears in the head of a rule in F then we say *p belongs to F*. If the predicate q appears in the body of a rule in F, but does not belong to F, then we say q is *used by F*. If an atom A has predicate p, and p belongs to F, then we may say that A also belongs to F. $\Box$

Recursiveness (as defined in Definition 5.2.1) is an equivalence relation between predicates. Every predicate has a unique minimal complete component to which it belongs. A program may be broken up into complete components according to the equivalence classes (called strongly connected components in [113]) induced on the predicates.

The minimal complete components have a natural relation associated with them: $F_1 \sqsubset F_2$ if some predicate belonging to F_1 is used by F_2. $\sqsubset$ must be an acyclic relation, since if $F_1 \sqsubset F_2 \sqsubset \cdots \sqsubset F_n \sqsubset F_1$ for some n, then none of $F_1, \ldots, F_n$ would be complete. We refer to $\prec$, the transitive closure of $\sqsubset$, as the *dependency relation* between components. $\prec$ is a partial order, with the property that a predicate belonging to a component F is defined in terms of predicates that either belong to F, or belong to a component F' where $F' \prec F$.

In what follows, when we refer to a component of a program, we mean a minimal complete component unless otherwise noted. Within this framework, a program is stratified if and only if none of its components contains a predicate that depends negatively on itself.

We must now consider what conditions must be placed on those components containing predicates depending negatively upon themselves. We want the well-founded semantics to be total, i.e., to make every ground literal either true or false. Weak stratifiability [81] is such a condition, and as we shall see, all modularly stratified programs are weakly stratified.

There is another important issue, namely the ordering of literals in the body of rules. As observed in Chapter 4, any top-down implementation of the well-founded semantics needs to expand negative literals in parallel so that we don't loop infinitely through negation when another subgoal fails.

Example 5.2.2: Consider the program

$$
\begin{array}{l}
p(X) \leftarrow t(X,Y,Z), \neg p(Y), \neg p(Z) \\
t(a,b,a) \\
t(a,a,b) \\
p(b)
\end{array}
$$

whose well-founded semantics is total, the (2-valued) model being $\{p(b), t(a,b,a), t(a,a,b)\}$. In fact, this program is weakly stratified.

Suppose we try to answer the query $? - p(a)$ top-down using a subgoal-at-a-time method. We assume left to right evaluation. We expand the first rule, using the first t rule for the first subgoal, yielding $\neg p(b)$ as the second subgoal. $p(b)$ succeeds, and hence $\neg p(b)$ fails, so we backtrack to the second rule for t. This gives $\neg p(a)$ as the second subgoal. We then try to expand $p(a)$ and fall into an infinite loop. Expanding the third subgoal before the second throughout won't help, since the second and third subgoals are symmetric. □

Expanding negative literals in parallel may generate unnecessary work, since if one of these subgoals fails the others are irrelevant.

For a magic set method to answer queries with negation efficiently bottom-up, the left-to-right[3] ordering of subgoals is crucial. In order to prevent the exploration of unnecessary parts of the search tree, we expand a subgoal only after we know that all subgoals appearing to its left have succeeded.[4] Example 5.2.2 demonstrates that there are weakly stratified programs for which any consistent sequential subgoal ordering will be impossible.

Following the arguments above, we want freedom from negative loops, and also subgoal-at-a-time evaluability. We will allow a component in which a predicate depends negatively on itself only if it satisfies the condition formalized below.

Definition 5.2.3: (Reduction of a component) Let F be a program component, and let S be the set of predicates used by F. Suppose S is fully defined by an interpretation M over the universe $\mathcal{U}$.

Form $I_{\mathcal{U}}(F)$, the instantiation of F with respect to $\mathcal{U}$, by substituting terms from $\mathcal{U}$ for all variables in the rules of F in every possible way. Delete from $I_{\mathcal{U}}(F)$ all rules having a subgoal whose predicate is in S, but which is false in M. From the remaining rules, delete all subgoals having predicates in S (which must be true in M) to leave a set of instantiated rules $R_M(F)$. We call $R_M(F)$ the *reduction of F modulo M*. □

Example 5.2.3: Let F be the component

$$\begin{aligned} p(X,Y) &\leftarrow t(X,Y,Z), \neg r(X), \neg q(Y,Z) \\ q(X,Y) &\leftarrow v(a,X,Y), p(Y,Y) \\ p(X,Y) &\leftarrow r(X), q(Y,X) \end{aligned}$$

and let M be the (2-valued) interpretation $\{t(a,d,e), t(b,c,f), r(b), v(a,b,c)\}$. Then $R_M(F)$ is

$$\begin{aligned} p(a,d) &\leftarrow \neg q(d,e) \\ q(b,c) &\leftarrow p(c,c) \end{aligned}$$

together with all possible instantiations of the rule

$$p(b,Y) \leftarrow q(Y,b)$$

with respect to the universe $\mathcal{U}$. □

[3] Other sideways information passing strategies (*sips*) that operate sequentially are equally valid. To guarantee good performance relative to top-down methods we construct the *sip* strategy for the magic set method to mimic the top-down order of expansion.

[4] The programmer places the "easier" subgoals to the left of the "harder" ones.

This definition of reduction is similar to the definition of reduction in [81] used to define weak stratifiability. There are two differences:

1. We restrict M to only those predicates used by F.
2. We do not delete instantiated clauses whose heads appear elsewhere in the program as unit facts.

We are now in a position to describe the class of programs we allow.

Definition 5.2.4: (Modular Stratification) Let $\prec$ be the dependency relation between components. We say the program P is *modularly stratified* if, for every component F of P,

1. There is a total well-founded model M for the union of all components $F' \prec F$, and
2. The reduction of F modulo M is locally stratified. □

Example 5.2.4: Let P be the program consisting of the component F from Example 5.2.3, together with some facts for the EDB predicates t, r, and v. Suppose that the given model M is the well-founded model for the EDB predicates. Then $R_M(F)$, as given in Example 5.2.3 is locally stratified, and so P is modularly stratified. Note that P itself is not locally stratified. □

Our definition of reduction also bears some similarity to the stability tranformation of Gelfond and Lifschitz [37] (see Definition 3.5.1). If the well-founded model for a component is total, then that model is also its unique stable model. The stability transformation effectively performs our iterated reduction in parallel, for all components at once. (This correspondence independently demonstrates part of Corollary 5.2.2 below.) However, not all programs with unique stable models are modularly stratified.

The definition of modular stratification is relative, in the sense that whether the reduction of a component is locally stratified depends on the truth values of the predicates defined at lower levels. For example, the game program from Example 5.2.1 above is modularly stratified if and only if m is acyclic. This property is unlike stratification, for example, where checking that a program is stratified can be done syntactically. We now show that modularly stratified programs are also weakly stratified [81].

Theorem 5.2.1: Every modularly stratified program is weakly stratified.
Proof: It is not difficult to show that a program P is weakly stratified if and only if, for every component F of P,

1. There is a total well-founded model M for the union of all components $F' \prec F$, and
2. The reduction of F modulo M is weakly stratified.

Since all locally stratified programs are weakly stratified [81], we can show by induction on the level of components that modularly stratified programs are weakly stratified. ∎

The converse of Theorem 5.2.1 is false, as illustrated by Example 5.2.2. Also, every stratified program is modularly stratified, as is every locally stratified program.

Corollary 5.2.2: Every modularly stratified program has a total well-founded model that is its unique stable model.
Proof: Since this property holds for weakly stratified programs [81]. ∎

To see how the well-founded model of a program may be composed from those of its components, recall that a locally stratified program has a unique perfect model [82] and hence a total well-founded model that coincides with the perfect model. The "lowest" components must be locally stratified; compute their well founded model M. The next lowest components are locally stratified when reduced modulo M; compute the well founded model of the reduced components and take the union with M. We can proceed in this way up the dependency relation between components until we have the well-founded model for the whole program.

When we look at a whole program, rather than at components, it may not be clear whether the program is modularly stratified. In general, for each new tuple added to a relation we would have to test all higher components for modular stratifiability.

Having to perform this test every time there is a change to the database is clearly undesirable. An alternative is to place constraints on the predicates used by a component. For example, the constraint for the game program of Example 5.2.1 would be simply that m is acyclic. Such a constraint would guarantee that the program is modularly stratified. When changes to m occur, we have to check only that acyclicity is not violated.

In fact, acyclicity is the typical example of a semantic constraint guaranteeing modular stratifiability. Such constraints are similar in nature to "monotonicity constraints" discussed in [15], which were used in the context of testing the termination of Datalog programs without negation. Placing constraints on components enables a modular approach to writing programs. The semantics of the game program is total as long as m is acyclic, independent of what particular constants appear as tuples of m. It is not necessary to look at all rule instantiations.

We shall show that deciding whether a function-free IDB is modularly stratified *for all possible EDB's* is undecidable in Section 7.2.3. Nevertheless, techniques such as monotonicity constraints provide sufficient conditions for modular stratifiability.

Example 5.2.5: Consider the following program component F, written in 1890, designed to identify those presidents of the United States who are even-numbered presidents.[5] (A president x is even-numbered if there have been an odd number of presidencies before x, where a presidency may last several consecutive terms.)

$$\mathit{even\_president}(X) \leftarrow \mathit{predecessor}(X,Y), \neg \mathit{even\_president}(Y)$$

$\mathit{predecessor}(X,Y)$ is an EDB relation which indicates that Y was in office immediately before X (even if X's presidency lasted several terms). Let M be the least model of the facts defining *predecessor*. Then assuming *predecessor* is acyclic we can show that $R_M(F)$ is locally stratified, since we can define a well-ordering $<_F$ on ground atoms such that $\mathit{even\_president}(Y) <_F \mathit{even\_president}(X)$ precisely when Y was a president some time before X. In this case the program is modularly stratified, and the well-founded semantics gives the desired results.

However, in 1893 Grover Cleveland was re-elected, having been previously defeated. Some problems then arose. $R_M(F)$ is no longer locally stratified, as *even_president(cleveland)* depends negatively on itself in $R_M(F)$. Further, depending on the timing of Cleveland's presidencies, the well-founded semantics may give unintended results.

If his first and second presidencies were even-numbered, then the well-founded semantics gives the expected results.[6] (Despite the loop, *even_president(cleveland)* is independently derivable.) However, if Cleveland's first presidency were odd-numbered, then the well-founded

[5] Note that this program is just a re-interpretation of the symbols in Example 5.2.1 above.

[6] In fact, this was the case; he was both the 22nd and 24th presidents.

semantics would make the atom *even_president(cleveland)* undefined, and would similarly make *even_president* undefined for all subsequent presidents. Finally, if Cleveland's first presidency was even and the second odd, then *even_president(cleveland)* would be true according to the well-founded semantics. Effectively, Cleveland's second presidency would be ignored and the parity of all presidents subsequent to his second term would be switched.

So how can we deal with this possibility? One solution is to associate with each president the period of office, so that two non-consecutive terms by a single president could be distinguished. The modified program would be

$$even\_president(X, P) \leftarrow predecessor(X, P, Y, Q), \neg even\_president(Y, Q)$$

where $predecessor(X, P, Y, Q)$ now means that X served during the interval beginning on date P, which was immediately after Y's presidency, which started on date Q. Let M be the least model of the rules for the new relation *predecessor*. We may now redefine the ordering $even\_president(Y, Q) <_F even\_president(X, P)$ to be true when P is temporally after Q. Since there have been only finitely many presidencies, the temporal order is a well-ordering. Hence our modified component F is such that $R_M(F)$ is locally stratified. □

Example 5.2.6: This example concerns the operation of a complex mechanism that is constructed from a number of components, each of which may itself have smaller components.[7] The mechanism is known to be *working* either if it has been (successfully) *tested*, or if all its components (assuming it has at least one component) are known to be *working*. We may express this in the following component F:

$$\begin{aligned}
&working(X) \leftarrow tested(X)\\
&working(X) \leftarrow part(X, Y), \neg has_suspect_part(X)\\
&has_suspect_part(X) \leftarrow part(X, Y), \neg working(Y)
\end{aligned}$$

Let M be the least model of the rules for *part* and *tested*. $R_M(F)$ is locally stratified if and only if *part* is acyclic. Acyclicity is a natural constraint, since a mechanism that was a sub-part of itself would presumably indicate a design error. Assuming *part* is acyclic, we may demonstrate that $R_M(F)$ is locally stratified according to the ordering defined by

$$working(X) <_F has\_suspect\_part(X)$$

for all X, and

$$has\_suspect\_part(X) <_F working(Y)$$

when $part(X, Y)$ holds. □

5.2.1 Modular Stratification from Left to Right

In a range restricted program, a variable appearing in a negative literal must appear somewhere in the body in a positive literal. Since the evaluation mechanism we propose operates left-to-right, we need some positive occurrence to occur to the left of the negative occurrence. Clearly any range restricted program can be transformed into a semantically equivalent one satisfying

[7]We adopt the convention that a mechanism is not a component of itself — we are only interested in smaller, simpler components.

this condition simply by placing some positive literal involving the variable X to the left of all negative literals involving X.

However, this is not enough to guarantee freedom from infinite loops through negation. We must examine the positioning of predicates used by the component relative to those belonging to the component. The predicates used by a component F give certain bindings for variables used elsewhere in the rule. If we intend to evaluate the rule from left to right, we must make sure that only the bindings (for predicates in the lower level model M) that "make $R_M(F)$ locally stratified" are passed to the negative subgoals whose predicates belong to F.

To see what can go wrong, consider the game program from Example 5.2.1 rewritten as follows with an additional unary predicate p that is true for any position.

$$w(X) \leftarrow p(Y), \neg w(Y), m(X,Y)$$

When evaluated by a top-down method from left to right, the program will loop infinitely through negation. However, the semantically equivalent program

$$w(X) \leftarrow p(Y), m(X,Y), \neg w(Y)$$

"behaves" when evaluated left to right.

The solution is to refine the notion of the local stratifiability of $R_M(F)$ to take account of the left-to-right order of evaluation.

Definition 5.2.5: A *rule prefix* is formed from a rule with n subgoals in the body by deleting the rightmost m subgoals, where $0 \leq m \leq n$. □

Definition 5.2.6: (Modular stratification from left to right) Let $\prec$ be the dependency relation between components. We say the program P is modularly stratified *from left to right* if, for every component F of P,

1. There is a total well-founded model M for the union of all components $F' \prec F$, and

2. The reduction of the set of all prefixes of rules in F modulo M is locally stratified. □

Example 5.2.7: Let P be the game program from above, containing the rule

$$w(X) \leftarrow p(Y), \neg w(Y), m(X,Y)$$

together with facts for p and m such that m is nonempty and acyclic, and p is true for all constants appearing as arguments of m. P is modularly stratified, but not modularly stratified from left to right. The rule prefix

$$w(X) \leftarrow p(Y), \neg w(Y)$$

will yield a violation of local stratifiability when X and Y are bound to the same variable. On the other hand, if the rule for w was rearranged to give

$$w(X) \leftarrow p(Y), m(X,Y), \neg w(Y)$$

then the program would be modularly stratified from left to right. □

Every program that is modularly stratified from left to right is also modularly stratified, and every modularly stratified program may be rearranged in such a way that it is modularly stratified from left to right, as shown below.

Lemma 5.2.3: Let P be a modularly stratified program. Then there is a re-ordering of the subgoals in the bodies of the rules in P such that the resulting program is modularly stratified from left to right.

Proof: For every rule r of the component F of P re-arrange the subgoals of r so that all sugoals whose predicates are used by F appear to the left of those subgoals whose predicates belong to F. Call the resulting program P'.

P' is modularly stratified, since P is modularly stratified. Suppose P' were not modularly stratified from left to right. Let F be a lowest component of P' whose set of rule prefixes is not locally stratified when reduced modulo M, where M is the well-founded model of the union of all components $F' \prec F$. (Such an M must exist since P' is modularly stratified.) Let F_P be the component F in P, before the subgoals were re-arranged.

Since all lower-component subgoals appear on the left in F, the rule prefixess r that have been reduced will satisfy the following property: The body of r is a subset of the body of a rule from the reduction of F_P modulo M (possibly re-arranged). Since F_P is locally stratified when reduced modulo M, we conclude that F is also locally stratified, contradicting our assumption. Hence P' is modularly stratified from left to right. ∎

Note that it is often possible to guarantee modular stratifiability from left to right without placing all lower-component subgoals at the left as in the proof of Lemma 5.2.3. On the other hand, rearranging the subgoals as above may cause a left-to-right evaluation method to flounder.

Example 5.2.8: Consider the program

$$\begin{array}{l} t(a,a) \\ p(X) \leftarrow t(X,X), \neg q(X) \\ q(X) \leftarrow t(X,Y), \neg t(Y,Z), p(Z) \end{array}$$

which is modularly stratified from left to right. A left-to-right expansion of the last rule will flounder due to the unbound variable Z in the subgoal $t(Y,Z)$. Placing the subgoal $p(Z)$ to the left of $t(Y,Z)$ would solve the floundering problem, but result in a program that was no longer modularly stratified from left to right. □

Unfortunately, we will not be able to handle programs that flounder when evaluated from left to right. We will restrict ourselves to "permissible" programs, as defined below.

Definition 5.2.7: A program P is *permissible* if it is range-restricted, modularly stratified from left to right, and if every variable appearing in a negative literal in the body of a rule in P also appears further to the left in a positive literal. □

At the expense of efficiency one could add an extra predicate that binds its argument to successive constants in the (finite) universe. By placing this predicate to the left of each floundering subgoal, one can prevent floundering without losing any expressive power. An alternative approach, which merits further research, would be to incorporate constructive negation [21, 84] into the evaluation mechanisms proposed in this chapter. Nevertheless, the author believes that programs that cannot be rearranged into permissible versions will be rare in practice.

Passing bindings from left to right is one of many possible "sideways information passing strategies" (sips) that could be used [13]. Alternative strategies can be put into our framework by applying Definition 5.2.6 with a refined notion of a rule prefix. For a subgoal S appearing in a rule r, the prefix of r corresponding to S is the sequence of subgoals (not necessarily to the left of S) from which bindings for S are generated, followed by S itself. Modifying the top-down and bottom-up methods, described in later sections, for arbitrary sips is beyond the scope of this thesis.

5.3 Top-Down Evaluation

5.3.1 Global SLS-Resolution

In this section we present a variant of global SLS-resolution from Chapter 4. In this chapter we will restrict ourselves to a left-to-right computation rule, so that the leftmost subgoal is always selected. This will allow us to compare bottom-up and top-down using the same order of evaluation. Although such a computation rule may not yield completeness in general, it will be sufficient in the context of permissible programs.

A negative subgoal flounders if it is selected and has an unbound variable appearing in it. In what follows we restrict ourselves to programs that do not flounder. Hence, for simplicity, we omit the case of floundered subgoals. In any case, global SLS-resolution is complete with respect to the well-founded semantics only for non-floundering programs.

Because we will be dealing with programs having no infinite recursion through negation, we may also simplify the construction by eliminating the global tree in favor of a simpler "negation tree." Negation trees are more convenient in our present context where we can expand negative literals ahead of positive ones. The redefinitions of "SLP-trees" and "global SLS-resolution" below apply only to this chapter.

Definition 5.3.1: (SLP-trees) Let P be a nonfloundering program, and let G be a goal. We define the SLP-tree T_G for G. The root node of T_G is G. If the goal Q' is any node of T_G then its children are obtained as follows:

- If Q' is empty, then we call it a *successful* leaf.
- Suppose that the leftmost literal L in Q' is positive. Let U_L be the set of program clauses whose heads unify with L. The children of H are obtained by resolving H with (a variant of) each of the rules in U_L over the literal L using most general unifiers. If there is no such K then H has no children, and is a *failed* leaf.
- Suppose the leftmost literal L in Q' is negative, say $\neg A$. (By our assumption about the absence of floundering, L must be ground.) Recursively construct the SLP-tree T_A for A.
  - If T_A is successful, then Q' is a *failed* leaf.
  - If T_A is failed, then Q' has a single child that is formed by deleting L from Q'.
  - Otherwise, T_A is an *indeterminate* leaf.

If T_G has a successful leaf, then T_G is *successful.* If every leaf of T_G is failed, then T_G is *failed.* Otherwise, T_G is indeterminate.

A *branch* of T_G is an acyclic path from the root of T_G. We associate with each successful leaf V an *answer substitution*, which is the composition of the most general unifiers used along the branch to V. □

Note that the definition of an SLP-tree itself is top-down, but that the status of the nodes as successful, failed or indeterminate is defined bottom-up. "Global SLS-resolution" is the process of determining whether a goal is successful, failed, or indeterminate, and if successful returning all answer substitutions.

Example 5.3.1: Consider the game program from Example 5.2.1 together with some move tuples, i.e.,

$$\begin{array}{l} w(X) \leftarrow m(X,Y), \neg w(Y) \\ m(a,b) \\ m(b,c) \end{array}$$

The SLP-tree for $w(a)$ is

$$\begin{array}{c} w(a) \\ | \\ m(a,Y), \neg w(Y) \\ | \\ \neg w(b) \end{array}$$

The SLP-tree for $w(b)$ is recursively constructed, and is

$$\begin{array}{c} w(b) \\ | \\ m(b,Y), \neg w(Y) \\ | \\ \neg w(c) \end{array}$$

The SLP-tree for $w(c)$ is recursively constructed, and is

$$\begin{array}{c} w(c) \\ | \\ m(c,Y), \neg w(Y) \end{array}$$

The leaf of the SLP-tree for $w(c)$ is failed, since there are no rule heads that unify with $m(c,Y)$. Hence $T_{w(c)}$ is failed, $T_{w(b)}$ is successful, and $T_{w(a)}$ is failed. □

Definition 5.3.2: We define the *negation tree* N_G for a goal G. The nodes of N_G are goals, and the root of N_G is G. Let H be any node of N_G. For every atom A for which T_A is recursively constructed in constructing T_H, A is a child of H. □

Example 5.3.2: For the program in Example 5.3.1, $N_{w(a)}$ is

$$\begin{array}{c} w(a) \\ | \\ w(b) \\ | \\ w(c) \end{array}$$

□

Note that every node except possibly the root of a negation tree is a ground atom. If the ground atom A is a child of H in a negation tree, there must be some branch in T_H to a node that has $\neg A$ as the leftmost literal. We now define a restricted version of global SLS-resolution from Chapter 4. We shall show that for the class of programs in which we are interested in this chapter, this simplified version is both sound and complete.[8]

Definition 5.3.3: (Global SLS-resolution) Global SLS-resolution is the top-down process of of finding all answer substitutions for a goal G by constructing the SLP-tree for G as in Definition 5.3.1. □

We will be particularly careful that the negation tree for every possible goal has finite depth. If every negation tree has finite depth then we know that every atom will either be successful or failed, and we will obtain a sound and complete implementation of the well-founded semantics. On the other hand, the program

$$p \leftarrow \neg p, q$$

has an infinite negation tree N_p. According to the definitions above, the SLP-tree for ?-p is indeterminate, even though p is false with respect to the well-founded semantics. The method presented above will loop infinitely through negation, clearly an undesirable situation. The rearranged version

$$p \leftarrow q, \neg p$$

has a finite negation tree for ?-p. Hence, the left-to-right ordering of subgoals will be crucial.

Theorem 5.3.1: Global SLS-resolution (as defined in Definition 5.3.3) is both sound and complete for non-floundering programs with finite-depth negation trees.
Proof: The proof follows from the corresponding results in Chapter 4. Since the negation tree is finite, there is no infinite recursion through negation, and so every goal is either successful or failed. ▮

To get soundness and completeness in general, one must use the more general method of Chapter 4, and avoid a purely left-to-right computation rule. In this paper we will look for sufficient conditions for negation trees to have finite depth, so that we do not need to consider indeterminate derivations. Permissibility is one such condition, as illustrated by the following theorem.

Theorem 5.3.2: If P is permissible then for every goal G, the negation tree N_G (with respect to P) will be of finite depth.
Proof: If P is permissible then it will not flounder, and so the method of Definition 5.3.3 is well-defined.

Consider an arbitrary branch B of the negation tree N_G. All nodes of the negation tree (with the possible exception of the root) along this branch are ground atoms, say $G_1, G_2, \ldots$ with $G_0 = G$. B is infinite if and only if there is some distinct pair of positions α and β containing the same atom, i.e., $G_\alpha = G_\beta$ for $\alpha > \beta > 0$. (Note that N_G cannot have infinite depth but only finite branches of unbounded length, since any branch longer than the number of atoms in the finite Herbrand base of the program must be infinite.)

[8] By "complete" we mean what some authors call "partially complete;" we do not require termination. A formulation of a top-down method that is guaranteed to terminate is presented in Section 5.3.2.

We prove the contrapositive of the theorem as stated. The proof is by induction on the components of P. Suppose the claim is true for the union U of all components F' of P such that $F' \prec F$. We show that the claim holds also for $U \cup F$.

Suppose that for some atom A belonging to $U \cup F$, N_A has an infinite branch. If A belongs to U, then by the induction hypothesis U is not modularly stratified from left to right, so neither is $U \cup F$. Suppose that there is no such atom belonging to U. Then there must be such an A belonging to F. Since N_A has an infinite branch, say B, there must be some ground atom A' that repeats on B. A' must belong to F.

Since all subgoals from U succeed precisely when they are true with respect to the well-founded semantics, by the induction hypothesis and Theorem 5.3.1, it follows that there is some cycle through negations in the prefixes of the rules in F reduced modulo the well-founded model for U. This cycle through negations implies that F is not modularly stratified from left to right. ∎

Theorem 5.3.2 does not hold for programs having function symbols, as illustrated by the program

$$\begin{array}{l} p(s(X)) \leftarrow t(X), \neg p(X) \\ t(0) \\ t(s(X)) \leftarrow t(X) \end{array}$$

for the goal $p(X)$.

Permissibility is not necessary to get finite negation trees, as illustrated by the program

$$p \leftarrow p, \neg p$$

for which the goal ?-p has an infinite branch in its SLP-tree, but a trivial negation tree.

Note that infinite positive derivations are considered failed under our approach. In order for global SLS-resolution to find all answer substitutions, and not get "lost" down an infinite (positive) branch of an SLP-tree, an appropriate method for searching SLP-trees is needed. A way of pruning infinite branches is discussed in the next section.

5.3.2 Top-Down with Memoing

Despite the soundness and completeness demonstrated in the previous section, global SLS-resolution may get lost down an infinite positive branch and hence not terminate. Several authors [30, 120, 121] have considered this problem in the context of Horn programs, and have proposed a form of memoing in which a positive literal is not "admissible" for selection if it is an instance of one of its ancestors.

Kemp and Topor [51] and, independently, Seki and Itoh [106] have generalized these proposals to the class of stratified programs. We extend these methods to a larger class of programs, namely those programs that have finite negation trees, and still retain soundness, completeness and termination properties. In particular, this method can be extended to permissible programs.

We now present our extension of the QSQR/SLS-procedure. While our method is similar to [51], there are several differences that we shall explain after giving the definitions. Our version of the QSQR/SLS procedure, given in Algorithm 5.3.1, is presented in such a way that the comparison with bottom-up evaluation will be clearer later on.

In the query evaluation procedure described below we will "memo" those facts that we have already derived so that we can re-use them in other parts of the computation. We memo all IDB predicates; in the terminology of [51], the set of "r-predicates" is the set of all IDB predicates.

Definition 5.3.4: A *lemma* is a literal having an IDB predicate. □

The state of the query answering procedure will be given by a set of lemmas together with a set of derivation-trees, i.e., a "derivation-forest." As the procedure proceeds, the set of lemmas will get larger, the trees will grow extra nodes, and new trees may be added to the forest.

Definition 5.3.5: A *derivation tree* is a tree whose nodes are goals. The root node of a derivation tree contains a single atom, and has an associated *polarity*, either positive or negative. Each non-root goal has an associated substitution and an associated rule. A *derivation forest* is a collection of derivation trees. □

Non-root goals G contain (partially instantiated) subgoals from the rule associated with G. New goals are formed by resolving the leftmost literal with a lemma, yielding a child goal (with one less literal) that has been instantiated according to the most general unifier used in the resolution. These details are explained in detail in Algorithm 5.3.1 below.

Definition 5.3.6: Suppose each rule r_j has s_j subgoals. For each rule r_j and for $i = 1, \ldots, s_j$, let $C_{j,i}$ denote the set of goals that have associated rule r_j, and whose leftmost literal is the ith in r_j. □

We now define the "depends positively," "depends negatively" and "settled" relationships. Conceptually, these relationships model whether the proof of a certain atom depends (positively or negatively) on the proof of another. For global SLS-resolution we would say that the atom P depends positively on the atom Q if the SLP-tree for P has a branch with leftmost literal Q. We would say that the atom P depends negatively on the atom Q if the negation tree for P had a child Q. The intuition behind these concepts is that a proof of $\neg P$ cannot be completed until everything that P depends negatively upon has been fully computed as true or false.

In the context of our top-down method with memoing, we modify the definition somewhat. Firstly, since we are only interested in proving $\neg P$ when the root has negative polarity, we shall restrict our "depends" relationships to have their first arguments appearing in roots of negative polarity. Secondly, since our memoing method eliminates duplicate derivations, it is possible that certain branches of a tree with a root of negative polarity have not been expanded because those branches were expanded in a tree with the same root but having opposite polarity. For example, if $p(a)$ and $\neg p(a)$ are subgoals, and $p(a)$ is reached before $\neg p(a)$, then the root of positive polarity will have all the children, and the root of negative polarity will have none. While this property does not affect correctness and eliminates duplicated effort, it also requires us to be careful in how we define the corresponding "depends" relationships.

Definition 5.3.7: Let P, Q, P', R, S, R' and S' be atoms, and let P and Q be ground. Let F be a derivation forest. We say P *depends positively* on R in F if either

- There is a tree in F with root P having negative polarity, and there is some rule r_j with head P' such that for some i there is a goal in $C_{j,i}$ that has leftmost literal R, associated substitution θ, and $P = P'\theta$, or

- For some S, P depends positively on S, there is some rule r_j with head S' such that for some i there is a goal in $C_{j,i}$ that has leftmost literal R', associated substitution θ, S and $S'\theta$ unify with most general unifier ϕ, and $R = R'\phi$.

We say P *depends negatively* on Q in F if either

- There is a tree in F with root P having negative polarity, and there is some rule r_j with head P' such that for some i there is a goal in $C_{j,i}$ that has leftmost literal $\neg Q$, associated substitution θ, and $P = P'\theta$, or
- For some S, P depends positively on S, there is some rule r_j with head S' such that for some i there is a goal in $C_{j,i}$ that has leftmost literal $\neg Q$, associated substitution θ, and S and $S'\theta$ are unifiable.

We say P is *settled* if some atom depends negatively on P, while either P or its complement is known, i.e., present as a lemma. □

We shall use the concept of depending negatively in order to determine when all "subsidiary" negative subgoals have been completed. Later, we shall define "depends negatively" in the context of bottom-up evaluation, and demonstrate an equivalence between that definition and Definition 5.3.7. Note that "depends negatively" here is a relationship between atoms, and is not the same relationship as that between predicates in Definition 5.2.1.

Definition 5.3.8: Let G' be a non-root goal with associated rule r_j. Suppose that the leftmost literal in G' was originally ith in r_j. Let θ be a substitution. We say ϕ is the *restriction* of θ to G' when ϕ is the subset of variable assignments in θ for variables in either the head of r_j, or in both a subgoal prior to the ith in r_j and a subgoal in r_j that is ith or later. □

We restrict substitutions so that differences in "irrelevant" variables do not prevent us from detecting that one goal subsumes another. The idea of restricting substitutions is similar to the way arguments of supplementary predicates are chosen for the magic sets method discussed later in Section 5.4.

We now define the QSQR/SLS procedure.

Algorithm 5.3.1: The algorithm of Figure 5.1 constitutes the QSQR/SLS procedure for the query ?-H where H is a literal, for a program P. If H is positive, let $H' = H$, otherwise let $\neg H' = H$. Note that if the program is range-restricted then all lemmas generated will be ground. We initialize the set of lemmas to contain those tuples in the EDB relations. When negative subgoals involving EDB predicates arise, rather than checking for the negative lemma explicitly, we shall simply check for the absence of the complement of the subgoal from the positive lemmas. □

0: Initialize the set of lemmas λ to the set of tuples in EDB relations, and the initial forest τ to contain the single goal H', of polarity equal to the sign of H.

repeat {

  repeat {

1a: For each root atom R for which the following was not performed on a previous iteration do {

  Let S be the collection of all rules whose heads unify with R. For each rule r_j in S resolve R with r_j in place, with most general unifier θ_j, to get a collection of goals G_j for each $r_j \in S$. Let ϕ_j be the restriction of θ_j to G_j. If ϕ_j is not subsumed by a substitution at any other node previously in $C_{j,1}$ then insert each G_j as a child of R, with associated rule r_j and associated substitution ϕ_j. }

1b: For each rule r_j do the following: Among those goals in $C_{j,1}$ that were just generated in step 1a, delete all but a set with most general associated substitutions.

2: $M := \emptyset$ /* M is the set of new positive lemmas */

For each empty goal G with substitution θ for which the following was not performed on a previous iteration do {

  Suppose the associated rule of G is r_j, with head Q. Add the lemma $Q\theta$ to M after renaming any variables in $Q\theta$ to new variables. }

Let M' be a set of most general lemmas in M. Add to λ all members of M' that are not already subsumed by lemmas in λ.

3: $N := \emptyset$ /* N is the set of literals for new root nodes */

For each nonempty non-root goal with leftmost literal L, and for which the following was not performed on a previous iteration do {

  If L involves an IDB predicate then add L to N. }

Let N' be a set of most general literals in N. Delete from N' any literal L whose atom is an instance of a root node already in τ of polarity equal to the sign of L. For each L in N' add the atom from L as a new root to the forest, with polarity equal to the sign of L, and with variables renamed to new variables.

4a: For each nonempty non-root goal G having selected literal L, for which the following was not performed on a previous iteration do {

  Suppose that L was originally the ith subgoal of r_j, where r_j is the associated rule of G. Let $\{L_1, \ldots, L_n\}$ be the lemmas that unify with L. Resolve L against each L_k, with most general unifier θ_k, to get a set of new goals G_k. If the substitution at G is ϕ, then the substitution at G_k is ψ_k, the restriction of $\phi\theta_k$ to G_k. Add as children of G those goals G_k whose associated substitution ψ_k is not subsumed by a substitution at any other node previously in $C_{j,i}$. The associated rule at G_k is r_j. }

4b: For each lemma L for which the following was not performed on a previous iteration do {

  Let $\{G_1, \ldots, G_n\}$ be nonempty non-root goals with respective selected literals $\{L_1, \ldots, L_n\}$ such that each L_k unifies with L. Suppose that each G_k has associated rule r_{j_k}, and that L_k was originally the i_kth subgoal of r_{j_k}. Resolve each L_k against L, with most general unifier θ_k to get new goals $G'_1, \ldots, G'_n$ respectively. If G_k has substitution ϕ_k, then the substitution at G'_k is ψ_k, the restriction of $\phi_k\theta_k$ to G'_k. Insert each G'_k as a child of G_k if ψ_k is not subsumed by a substitution at any other node previously in C_{j_n,i_n}. The associated rule at G'_k is r_{j_k}.}

4c: For each rule r_j and for $i = 1, \ldots, s_j$ do the following: Among those goals in $C_{j,i}$ that were just generated in 4a and 4b, delete all but a set with most general associated substitutions.

5: Infer new "depends positively," "depends negatively" and "settled" relationships.

  } **until** no changes occur

6: For each root R of negative polarity such that R is not present as a lemma, and such that every ground atom Q that R depends negatively upon is known to be settled after step 5 above, add $\neg R$ as a lemma (unless it is already present).

} **until** no changes occur

7: If H is positive then output all lemmas for H, or "no" if there are none. If H is negative then output H if H is a lemma and "no" otherwise. $\square$

Figure 5.1: Algorithm 5.3.1: The QSQR/SLS procedure.

Example 5.3.3: Let P be the program

$$p(X) \leftarrow b(X)$$
$$p(X) \leftarrow e(X,Y,Z), \neg p(Z), p(Y)$$

together with a set of facts for the EDB predicates b and e. To make the example more concrete, think of X, Y and Z as integers from some range $1, \ldots, n$, where $e(X,Y,Z)$ is true precisely when Y and Z are proper factors of X, i.e., $YZ = X$ and both $Y < X$ and $Z < X$. (The reduction of this program component is locally stratified since $e(X,Y,Z)$ implies $Y < X$ and $Z < X$; hence the program is modularly stratified for this e.)

Suppose $b(X)$ is true precisely when X is prime. Then it is not difficult to verify that $p(X)$ is true (according to the well-founded semantics) when X is the product of an odd number of primes, and false otherwise.

Consider how QSQR/SLS-resolution would answer the query ?–$p(18)$. The forest constructed by the QSQR/SLS method is given in Figure 5.2. The IDB lemmas inferred would be $p(2)$, $p(3)$, $p(18)$, $\neg p(6)$ and $\neg p(9)$. Note that the trees with roots of negative polarity are not expanded because the expansion has been done in the corresponding trees of positive polarity. The $\bullet$ symbol indicates an empty goal node. $\Box$

The main difference between SLP-trees and derivation-trees generated by the QSQR/SLS procedure is the resolution with lemmas. Any atom that is an instance of an atom that has appeared previously can only be resolved with lemmas and not rules. Infinite branches are effectively pruned.

While we have not specified exactly how new "depends positively," "depends negatively," and "settled" relationships are inferred at step 5, these relationships can be computed using techniques similar to those in step 4. We omit the details here.

We now explain the major differences between Algorithm 5.3.1 and the algorithms presented in [51] and [106].

The first difference is in how the lemmas are established. In [51], "proof-segments" of SLD-type trees are used rather than derivation trees. In [106] they are called "subrefutations." The definition of proof segments is complicated by the fact that subsidiary information about other subgoals is embedded in the trees. In our case, every derivation tree contains only information relevant to the subgoal at the root of the tree. When a subgoal is first encountered by Algorithm 5.3.1 in a nonroot node of some derivation tree, a new derivation tree is created with that subgoal as root. The original derivation tree proceeds only by using the lemmas generated by the new tree. While this difference is inessential in as far as what is computed, having a new tree for each atom that is memoed makes the procedure easier to understand, and also provides an easy mechanism by which subsumption of root nodes can be checked.

Kemp and Topor consider two admissibility tests. The less redundant of the two tests compares the selected atom against all previously visited atoms *in the current tree.* This test can lead to significant redundant computation in subsidiary trees. However, Kemp and Topor do suggest ways of improving the admissibility test by memoing negative subgoals, and subgoals expanded at lower levels. Our notion of admissibility incorporates these extensions. While we do not mention admissibility explicitly, our admissibility test is built-in to part 3 of Algorithm 5.3.1.

Another difference is the way subsidiary trees are expanded. By maintaining information about which predicates depend negatively on others, we can wait until all subsidiary goals have been completed before completing the current goal. Kemp and Topor achieve the same effect by

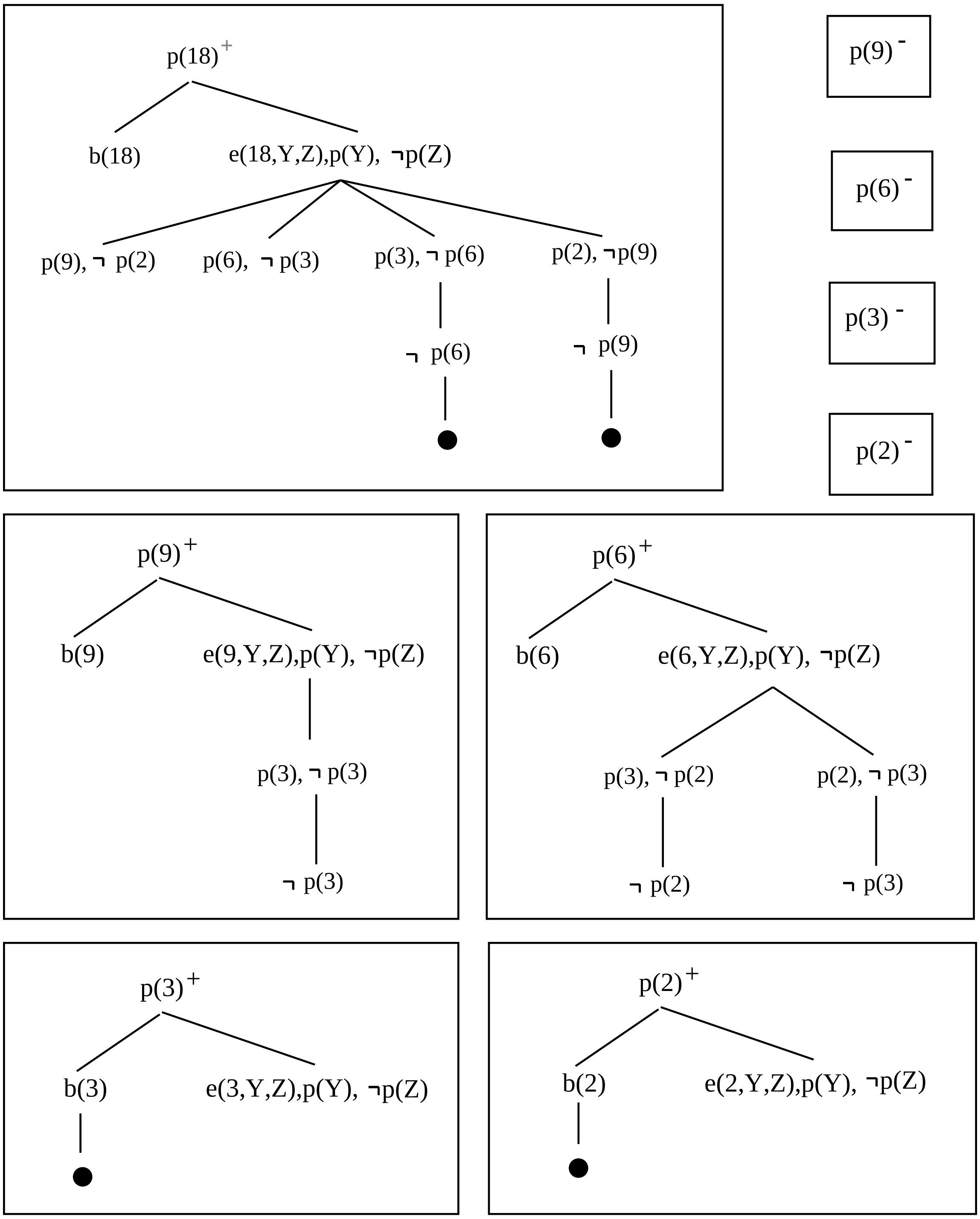

Figure 5.2: Forest generated in Example 5.3.3.

suspending the local computation and making a recursive call to expand a lower level subgoal, and continuing only when that call has terminated. Our dynamic approach has the advantage that only those subgoals that do depend negatively on lower-level subgoals are suspended, while other subgoals can be expanded during that time.

Unlike [51] and [106], our method distinguishes between EDB subgoals and IDB subgoals, using simple look-ups for EDB subgoals rather than performing resolution on them.

A more technical difference occurs in step 3 of our algorithm. In [51] and other similar work on top-down methods with memoing [105, 106] it is possible to generate a subquery at a certain point, and at a later point in the same round generate a more general subquery, in which computation is repeated. For example, suppose we encounter the selected atom $p(X, X)$. At a later point on the same round we encounter $p(X, Y)$, which subsumes $p(X, X)$. Previous approaches would create a tree for $p(X, X)$ and then create a second root for $p(X, Y)$. They would keep the less general goal if it happened to be considered before the more general goal. Our algorithm eliminates this redundancy since only the most general new subgoals are expanded into trees.

It may appear that there is another difference in the way we establish lemmas, by inferring instances of the head atom from a rule rather than instances of the root atom in a tree. This apparent difference is inconsequential: since the root and the head are unified, we may infer instances of either. However, our method restricts substitutions so that irrelevant intermediate variables do not have to be carried around. This restriction has benefits for discovering redundant derivations in step 4, and requires that we infer instances of the atom in the rule head rather than instances of the root atom.

Our method can handle all programs with finite negation trees, rather than only stratified programs. In fact, the proofs of soundness, completeness and termination from [51] extend in a straightforward way to programs with finite negation trees. The essential part of the proofs in [51] is an inductive argument on the number of levels guaranteed by the stratification. The only change to the proofs to reach the larger class is to proceed by induction on the height of the negation tree. If the negation tree is finite, then the proofs succeed. The differences outined above do not affect correctness; they show that in some cases the method proposed in [51] performs more (redundant) computation than ours.

Theorem 5.3.3: (Correctness) Let P be a range-restricted nonfloundering program having finite negation trees. Let L be a literal, (ground if negative), and let the query be ?-L. Then Algorithm 5.3.1 terminates such that for every ground substitution θ for L:

$$L\theta \text{ belongs to the (total) well-founded model for } P \text{ if and only if } L\theta \text{ is output.}$$

Proof: The proof is similar to that in [51], with a proof by induction on the finite number of levels guaranteed by the finite negation tree, rather than on the levels generated by a stratification. The comparison is with global SLS-resolution rather than with SLS-resolution of [85] since the latter is well-defined only for locally stratified programs. We implicitly use the soundness and completeness of global SLS-resolution with respect to the well-founded semantics. ∎

5.4 Bottom-Up Evaluation

It is desirable to have a bottom-up alternative to global SLS-resolution, which is top-down. Bottom-up methods have the potential to work in a relation-at-a-time fashion, using efficient methods to compute large joins. See [113] for further discussion on the relative merits of top-down versus bottom-up methods.

5.4.1 Naive and Semi-Naive Evaluation

"Naive" and "semi-naive" bottom-up evaluation of rules for programs without negation are now standard concepts in deductive databases [113]. While we shall rely on semi-naive evaluation, there is one subtlety that arises due to the presence of variables. When tuples can have variables in them[9] we must perform something more sophisticated than duplicate elimination. Consider two possible approaches to duplicate elimination, which we term the "incremental" and "global" approaches. Suppose we have a set of (possibly nonground) tuples S, and wish to add some tuples from a set T to S.

The incremental approach may be described as follows. For every element t of T in some nondeterministic order, test whether t is subsumed by some element already in the current version of S, and if not then add t to S. The result of this operation may depend on the order in which tuples from T are selected. Once a tuple is in S it will never be removed.

The global approach is to form $S' = S \cup T$ and to keep in S' a minimal set of most general tuples from S'. The result of this operation is unique up to renaming of variables.

The incremental approach requires approximately $|S||T| + |T|^2$ matching operations. The global approach requires approximately $2|S||T| + 2|T|^2$ matches, since subsumption must be tested in both directions.

In this paper we take an intermediate approach to duplicate elimination. Our duplicate elimination strategy, which we call the "regional" approach is to discard all but the most general elements of T, and then add to S all those members of T that are not subsumed by members of S.

The result of regional duplicate elimination does not depend on the order of adding elements from T to S, while it requires approximately $|S||T| + 2|T|^2$ matches. In our context, S will be large compared to T, since S represents all information derived so far while T represents information derived at the current step. Hence regional evaluation has cost approximately the same as incremental evaluation, and about half that of global evaluation.

Regional evaluation has an additional benefit over incremental evaluation in our context, where tuples may be used to generate new tuples by evaluating rules. Consider the following example.

Example 5.4.1: Suppose we are firing the following rule using semi-naive evaluation.

$$p(X,Y) \leftarrow q(X,Y), r(Y,Z)$$

Firstly, suppose $q(a,a)$ is known, and the more general fact $q(W,W)$ is derived at a later stage. Neither incremental nor regional evaluation will eliminate the tuple $q(a,a)$; deleting $q(a,a)$ at this point will obviously make no difference to the prior execution.

On the other hand, suppose we have a new r tuple $r(V,V)$ and we wish to fire the rule above. We may choose to join $r(V,V)$ with $q(a,a)$ first yielding $p(a,a)$, and then with $q(W,W)$ yielding $p(W,W)$. However, since the less general tuple is generated first, the duplication persists in p if we use incremental duplicate elimination, and may be propagated through further rules to

[9] While our answers will always be ground since the program is assumed to be range restricted, some of the intermediate predicates we shall use may have nonground tuples.

other relations. On the other hand, the regional approach deletes the less general tuples among those newly derived, thus preventing duplication from propagating unnecessarily through the database. □

Finally, since the particular names of variables in nonground tuples should not matter, we shall assume that all variables in nonground tuples are renamed so that none are shared by more than one tuple. By renaming we avoid problems like using the tuples $p(X)$ and $q(X)$ with the rule

$$r(Y,Z) \leftarrow p(Y), q(Z)$$

to generate $r(X,X)$ rather than the more general $r(S,T)$.

5.4.2 Bottom-Up with Negation

A naive implementation of the bottom-up computation of the original program will result in many irrelevant tuples being computed. Top-down can beat bottom-up evaluation of the *original* program because it can restrict the computation to only those tuples relevant to the query. This problem is now well-understood for programs without negation, and several techniques have been proposed [12]. One of these is magic sets [11, 13, 88], which rewrites the program in such a way that the binding information that would have been passed down by a top-down method is incorporated into the bottom-up evaluation. In fact, for Datalog programs without negation, semi-naive bottom-up evaluation of the rewritten program performs at least as well as a straightforward top-down method [112].

In Section 5.4.3 we describe a magic sets method for evaluating modularly stratified programs bottom-up. In particular, this method also works for stratified programs. Our proposal extends the magic templates method of [88] rather than the original magic sets proposals.

Before we introduce the magic sets transformation, we describe a transformation of the program whose bottom-up evaluation is sound and complete with respect to the well-founded semantics for nonfloundering programs with finite negation trees, and in particular, for permissible programs. This transformation results in a program that is not Datalog. Several new constructs are necessary in order to be able to tell when we know "everything we need about p" so that instances of $\neg p$ can be correctly inferred.

The essence of this transformation is having the ability to identify when an atom depends negatively on another. If an atom a depends negatively on an atom b, then we must be sure that b is fully evaluated before we can say anything about a (unless a is independently provable). On the other hand, if a does not depend negatively on anything, and if a has not been derived when the iteration reaches a fixpoint, then it is safe to infer $\neg a$.

Note that inferring $\neg a$ may allow other rules to fire, and so another round of iteration may be required. When we reach a fixpoint at which no new negative information is inferred, we can terminate the iteration. We may call this end result the *final* fixpoint, and the other fixpoints *intermediate*; where the distinction is not crucial we will just use the term *fixpoint*.

The new constructs are as follows:

1. An extra modal operator $\lhd$. $\lhd p(a)$ holds if and only if, at the most recent fixpoint, $p(a)$ had been deduced.

2. Meta-variables, usually denoted by Q and R, that may unify with any atom.

3. The *depends* predicate (which we abbreviate to d), that maintains a record of which atoms depend negatively on which other atoms.

4. The *un-depends* predicate (which we abbreviate to d'), that maintains a record of those dependencies that have been finished with. For example, if $d(p,q)$ holds, and we find q is true, then $d(p,q)$ is no longer relevant and we assert $d'(p,q)$.

5. The *currently-depends* predicate (which we abbreviate to dd), which is precisely the relation defined by the difference $d - d'$.

Definition 5.4.1: (Negation Replacement) Let r be a rule. The *negation replacement* of r is formed from r as follows:

- Replace every negative IDB subgoal $\neg q(\vec{X})$ by the subgoal $\Box q(\vec{X})$, which is an abbreviation for the conjunction $\neg \lhd\ dd(q(\vec{X}), \_), \neg \lhd\ q(\vec{X})$.
- Replace every negative EDB subgoal $\neg q(\vec{X})$ by the subgoal $\sim q(\vec{X})$.

$\Box$

Informally, $\Box Q$ means that, at the most recent fixpoint, Q did not depend negatively on anything, and Q had not been computed. $\sim Q$ holds when Q is not true at the present time; since the extension of EDB predicates does not change throughout the computation we can use the simpler form of negation for them. Also, $dd(X,Y)$ may be seen as an abbreviation for $d(X,Y), \sim d'(X,Y)$.

Definition 5.4.2: (Well Founded Rewriting) Let P be a Datalog program. We construct the *well-founded rewriting* of P, denoted *WFR(P)*, as follows:

1. For every rule r of P the negation replacement of r is in *WFR(P)*.
2. For every rule $p(\vec{X}) \leftarrow q_1(\vec{X}_1), \ldots, q_n(\vec{X}_n)$ of P, and for each i from 1 to n,
   - if $q_i(\vec{X}_i)$ is negative, say $\neg t_i(\vec{X}_i)$, then the negation replacement of the rule
   $$d(p(\vec{X}), t_i(\vec{X}_i)) \leftarrow q_1(\vec{X}_1), \ldots, q_{i-1}(\vec{X}_{i-1})$$
   is in *WFR(P)*.
   - if $q_i(\vec{X}_i)$ is positive, and q_i is an IDB predicate then the negation replacement of the rule
   $$d(p(\vec{X}), Q) \leftarrow q_1(\vec{X}_1), \ldots, q_{i-1}(\vec{X}_{i-1}), dd(q_i(\vec{X}_i), Q)$$
   is in *WFR(P)*.
3. The following two rules are in *WFR(P)*:
   - $d'(Q,R) \leftarrow dd(Q,R), R$
   - $d'(Q,R) \leftarrow dd(Q,R), \Box R$
4. Nothing else is in *WFR(P)*. $\Box$

We present an example, and then discuss the correctness of this transformation.

Example 5.4.2: Consider the program from Example 5.3.3 for which $p(X)$ is true when X is the product of an odd number of primes.

The well-founded rewriting of P consists of the rules

$$p(X) \leftarrow e(X, Y, Z), \Box p(Z), p(Y) \tag{1}$$
$$p(X) \leftarrow b(X) \tag{2}$$
$$d(p(X), p(Z)) \leftarrow e(X, Y, Z) \tag{3}$$
$$d(p(X), Q) \leftarrow e(X, Y, Z), \Box p(Z), dd(p(Y), Q) \tag{4}$$
$$d'(Q, R) \leftarrow dd(Q, R), R \tag{5}$$
$$d'(Q, R) \leftarrow dd(Q, R), \Box R \tag{6}$$

together with the original e and b facts.

Consider the behavior of the rewritten program under bottom-up evaluation. The first, fourth and sixth rules above will not contribute until after the first fixpoint is reached, since the operator $\Box$ appears in their bodies.

The second rule adds the primes to p, while the third rule will derive $d(p(X), p(Z))$ (and hence also $dd(p(X), p(Z))$) precisely when Z is a proper divisor of X. The fifth rule will add $d'(p(X), p(Z))$ (and hence delete $dd(p(X), p(Z))$ from dd) when Z is prime (whence $p(Z)$ holds).

At this point, no further rules can fire, and so we reach a fixpoint. Now, the rules with $\Box$ in their bodies may fire. The first rule fires when Z has only prime proper factors (i.e., is the product of two primes) and when Y is prime, deducing $p(X)$ for values of X that are the product of three primes. These new values may be fed back into the first rule to get $p(X)$ for X being the product of five primes, and so on.

The second and third rules can't add anything new, and the fourth rule repeats values that were generated before the previous fixpoint. The fifth rule adds $d'(p(X), p(Z))$ for all Z that are the product of an odd number of primes. The sixth rule adds $d'(p(X), p(Z))$ for all Z that are the product of two primes. At this point, the second intermediate fixpoint is reached.

At successive fixpoints, $d'(p(X), p(Z))$ will be added by the sixth rule for Z being products of successive even numbers of primes. The final fixpoint occurs once d' catches up to d. This information is summarized in Figure 5.3. $\Box$

There are two possible objections to using a bottom-up evaluation of *WFR(P)* as a mechanism for evaluating queries. First is the observation that many irrelevant tuples are generated. In Section 5.4.3 we present a magic-set modification of *WFR(P)* that prevents the generation of irrelevant information. The second objection is that there may be substantial overhead involved in maintaining the d and d' predicates. We will discuss this issue in Section 5.4.4.

Note that *WFR(P)* may itself be not range restricted (although it cannot flounder). For example, if P contains the rule

$$p(X, Y, Z) \leftarrow t(X, Y), u(Y, Z)$$

then *WFR(P)* contains the rule

$$d(p(X, Y, Z), Q) \leftarrow d(t(X, Y), Q)$$

$p(X)$	$\Box p(X)$	$d(p(X), p(Z))$	$d'(p(X), p(Z))$
X prime		Z divides X	Z prime and Z divides X
X product of an odd number of primes	X product of two primes		Z product of an odd number of primes, or of two primes, and Z divides X
	X product of four primes		Z product of four primes and Z divides X
	X product of six primes		Z product of six primes and Z divides X
	$\vdots$		$\vdots$

Figure 5.3: Order of inference. Each horizontal line corresponds to a fixpoint.

in which Z does not appear in the body. However, such violations of range restrictedness occur only in the first argument of the predicate d (and hence also dd and d'). If $dd(p(\vec{X}), Q)$ holds, and $\vec{X}$ is more general that $\vec{a}$, then we say that $dd(p(\vec{a}), Q)$ also holds.

Lemma 5.4.1: (Church-Rosser Property) Let I be any fixpoint during the bottom-up evaluation of *WFR(P)*. Then the extension of dd at I is independent of the order in which rules are fired.
Proof: If no d' tuples are generated in the computation between the previous fixpoint and I, then dd grows monotonically, and hence the order of firing is unimportant. Suppose that $d(a, b)$ holds, and that $d'(a, b)$ is generated between the previous fixpoint and I. The only way that the evaluation of *WFR(P)* could depend on the order of evaluation could be if for some such a and b, $dd(a, b)$ caused a tuple to be generated that would not be present if $dd(a, b)$ did not hold. However, by the construction of *WFR(P)*, all tuples generated by rules having $dd(a, b)$ as a subgoal will be of the form $dd(Q, b)$. Since the reason for generating $d'(a, b)$ must be dependent only upon b, all such $d'(Q, b)$ will be subsequently generated, and so at I, dd will be independent of the order of firing of the rules. ∎

Corollary 5.4.2: The extent of all predicates in every fixpoint is independent of the order in which rules are fired. ∎

We prove the correctness of the bottom-up evaluation of *WFR(P)* with respect to the well-founded semantics for nonfloundering programs with finite negation trees. In particular, the correctness theorem below applies to all permissible programs.

Theorem 5.4.3: If P is a nonfloundering program having a finite negation tree, then the bottom-up evaluation of *WFR(P)* generates exactly the true atoms in the (two-valued) well-founded model of P.
Proof: The proof will be by induction, comparing the evaluation of *WFR(P)* with an "implementation" of global SLS-resolution from Section 5.3.1.

Our implementation[10] of global SLS-resolution is to repeat the following sequence of steps for *every ground atom* A in the Herbrand base of P until nothing more can be done:

1. Let N be the set of trees that are not currently labelled *failed*, but whose leaves are all *failed*. Label each member of N as *failed*. (The first time this statement is executed it will have no effect.)

2. Construct the SLP-tree T_A according to Definition 5.3.1 as completely as possible, but without labelling any trees as *failed*. If the status of T_B is needed to determine the status of a node L, and the status of T_B is not yet known, then nothing is done with L (until later, once the status of T_B is resolved).

Suppose that the procedure above is iterated n times before nothing new is generated. We shall show that at the end of each iteration i where $1 \leq i \leq n$, the following correspondence holds between our implementation of global SLS-resolution and the bottom-up evaluation of *WFR(P)* at the ith fixpoint. For all ground atoms A and B:

	Global SLS-resolution	*WFR(P)*
(a)	T_A is successful	A is inferred
(b)	T_A is failed	$\Box A$ holds
(c)	There is an atom B such that T_B is neither successful or failed, and $\neg B$ is the leftmost atom in a node of T_A.	$dd(A, B)$ holds

Base Case. After one iteration, no tree is labelled *failed*, but all positive resolutions have been performed. Similarly $\Box S$ holds for no S since there is no previous fixpoint.

(a) If T_A is successful, then it has a successful branch D without any negative subgoals occurring. By iterating the rules corresponding to those used on D, the bottom-up computation will generate A. (Since the program is range-restricted, it won't generate something more general than A.) Conversely, suppose that A is inferred. Then the rules used in the bottom-up evaluation are negation-free, and can be used in the top-down resolution to generate a successful leaf in T_A.

[10] While we call this procedure an implementation, we should point out that it is not, in general, effective. In particular, infinite positive branches in SLP-trees may be constructed. However, the soundness and (partial) completeness of global SLS-resolution will guarantee the soundness and (partial) completeness of the evaluation of *WFR(P)*. We shall make a separate argument for the termination of the latter.

(b) Trivial, since no tree is failed, and no tuples $\Box S$ hold.

(c) Suppose that there is an atom B such that T_B is neither successful or failed, and $\neg B$ is the leftmost atom in a node M of T_A. Denote the branch from A to M by D. Let the sequence of rules used in the resolution to M be $r_1, \ldots, r_m$, and suppose $\neg B$ appears in the body of r_j where $1 \leq j \leq m$. Let C be the head of r_j. Every subgoal to the left of $\neg B$ in r_j must be positive.

By the construction of *WFR(P)* and by part (a) above, $d(C, B)\theta$ holds for θ corresponding to the composition of most general unifiers used on D from r_{j+1} to r_m. Suppose that the literal C', with which C was resolved using rule r_j, was introduced in rule $r_{j'}$, where $j' < j$. (If C' was the root A, then clearly $d(A, B)$ holds.) Let the head of $r_{j'}$ be C''. By the construction of *WFR(P)*, $d(C'', B)\theta'$ holds, where θ' is the composition of most general unifiers used from $r_{j'+1}$ to r_m. We can continue this process up the branch until we eventually reach the root A, thus demonstrating that $d(A, B)$ holds. Since B is neither successful nor failed, neither B nor $\Box B$ hold (the latter trivially), and so $d'(A, B)$ cannot have been inferred. Hence $dd(A, B)$ holds at the first fixpoint.

Conversely, suppose $dd(A, B)$ holds for some ground tuples A and B. Then $d(A, B)$ holds, but $d'(A, B)$ does not. Because $d'(A, B)$ does not hold, it follows that neither B nor $\Box B$ is true (the latter trivially), and so by part (a) T_B is neither successful nor failed.

By tracing the rewritten rules used to derive $d(A, B)$, we can construct a branch in T_A of the corresponding original rules, yielding a node with leftmost literal $\neg B$. By the construction of *WFR(P)*, all rewritten rules used to derive $d(A, B)$ cannot have any subgoals of the form $\Box S$, and so we can use part (a) above to construct the corresponding branch.

Induction Step. Suppose that the claim holds for iterations up to $i - 1$. We shall show that the claim holds for i.

(a) Similar to part (a) of the base case above, except that we use the induction hypothesis for negative subgoals for the correspondence between trees T_A that are failed and tuples $\Box A$.

(b) Suppose T_A is shown to be failed at stage i. Then at some point during the previous iteration the tree T_A became completely expanded, and all its leaves were failed at the end of the $i - 1$st iteration. Since all its leaves were failed at this step, T_A is not successful after iteration $i - 1$, and also there is no node in T_A with a leftmost literal $\neg B$ such that the status of T_B as either successful or failed is unknown. By the induction hypothesis and by the definition of $\lhd$, at step i both $\neg \lhd dd(A, \_)$ and $\neg \lhd A$ hold, and hence $\Box A$ holds.

Conversely, suppose $\Box A$ is inferred at step i. Then at step $i-1$, $dd(A, B)$ did not hold for any B and A did not hold. By the induction hypothesis, T_A is not successful, and there are no nodes in T_A with leftmost literal $\neg B$ such that the status of T_B as either successful or failed is unknown. By the definition of global SLS-resolution, T_A will be labelled failed at the ith iteration.

(c) Similar to part (c) of the base case above, except that we use the induction hypothesis for negative subgoals for the correspondence between trees T_A that are failed and tuples $\Box A$.

In particular the correspondence holds at the final fixpoint, at which time all SLP-trees are fully constructed, and the evaluation of *WFR(P)* is complete. Soundness and (partial) completeness then follow from Theorem 5.3.1. Since only finitely many tuples can be generated in the evaluation of *WFR(P)*, it follows that this bottom-up evaluation terminates. ▮

To see where this method breaks down when the program does not have a finite negation tree, consider the weakly stratified program from Example 5.2.2. The first rule yields the rewritten rules

$$\begin{array}{c} p(X) \leftarrow t(X,Y,Z), \Box p(Y), \Box p(Z) \\ d(p(X),p(Y)) \leftarrow t(X,Y,Z) \\ d(p(X),p(Z)) \leftarrow t(X,Y,Z), \Box p(Y) \end{array}$$

from which it is possible to derive $d(p(a),p(a))$ given the relation for t. $p(a)$ will never be derived, since (as can easily be shown) the method is sound with respect to the well-founded semantics for all programs. Hence $d'(p(a),p(a))$ will never be derived, since $p(a)$ can't be derived, and $\Box p(a)$ requires $dd(p(a),\_)$ to be false. Thus $\Box p(a)$ will never be derived, and the method is incomplete.

The same problem occurs for programs that, while modularly stratified, are not modularly stratified from left to right, as in Example 5.2.7. While the program of Example 5.2.7 can have its subgoals rearranged so that it becomes modularly stratified from left to right, there is no way to perform a similar rearrangement for Example 5.2.2.

5.4.3 Magic Sets

Recall how QSQR/SLS-resolution answered the query ?–$p(18)$ from Example 5.3.3. Bindings for the variables are passed down from parent node to child node, thus restricting the search process to only those numbers that are factors of 18.

By comparison, a direct bottom-up evaluation of the program as in the previous section would be relatively inefficient for deciding whether $p(18)$ was true or false. It will decide whether $p(N)$ or $\neg p(N)$ holds for *all* numbers N in the range $1,\ldots,n$.

We shall present a magic set rewriting of *WFR(P)* that significantly reduces the amount of work that needs to be done. Using magic predicates in the body restricts the tuples generated during the bottom-up computation to those that are relevant. We use a version of the magic templates method [88] that is also described in [113]. For every predicate p in the program we introduce a new "magic" predicate m_p. Magic predicates have the same number of arguments as the original predicates.

The rewriting of *WFR(P)* will have magic predicates in the bodies of the rules, and will thus restrict the set of tuples generated to those we are interested in, as we become interested in them. (We become interested in a subgoal when all the subgoals to its left in a rule whose head we are interested in are true.) The magic predicates, like the ordinary predicates, may grow after the first intermediate fixpoint. Hence we should not infer $\Box p(\vec{X})$ unless $\vec{X}$ was in the magic set for p at the previous fixpoint. For this reason, $\Box p(\vec{X})$ will denote the conjunction

$$\lhd\, m\_p(\vec{X}), \neg \lhd\, dd(p(\vec{X}),\_), \neg \lhd\, p(\vec{X})$$

in the magic-rewritten version of *WFR(P)*. In other words, $\Box p(\vec{X})$ holds if, at the most recent fixpoint, $\vec{X}$ was in the magic set for p, $p(\vec{X})$ did not depend negatively upon anything, and $p(\vec{X})$ had not been computed. Had we omitted the $\lhd\, m\_p(\vec{X})$ conjunct, we would have inferred $\Box p(\vec{X})$ simply because it was not yet "relevant." Since it may become relevant at a later stage, such inference is clearly undesirable.

Definition 5.4.3: (*MWFR(P)*) Let P be a program, and *WFR(P)* its well-founded rewriting. We form *MWFR(P)*, the magic-set rewriting of *WFR(P)* as follows:

1. For every rule r in *WFR(P)*, with head of the form $p(\vec{X})$, prepend the subgoal $m\_p(\vec{X})$ to the body of r.

2. For every rule r in *WFR(P)*, with head of the form $d(p(\vec{X}), Q)$, prepend the subgoal $m\_p(\vec{X})$ to the body of r.

3. Every rule in *WFR(P)* with head predicate d' is also in *MWFR(P)*.

4. For every rule of *WFR(P)* of the form $p(\vec{X}) \leftarrow q_1(\vec{X}_1), \ldots, q_n(\vec{X}_n)$ and for each i from 1 to n,

   - if $q_i(\vec{X}_i)$ is negative, say $\Box t_i(\vec{X}_i)$, and t_i is an IDB predicate, then the rule
   $$m\_t_i(\vec{X}_i) \leftarrow m\_p(\vec{X}), q_1(\vec{X}_1), \ldots, q_{i-1}(\vec{X}_{i-1})$$
   is in *MWFR(P)*.
   - if $q_i(\vec{X}_i)$ is positive, and q_i is an IDB predicate, then the rule
   $$m\_q_i(\vec{X}_i) \leftarrow m\_p(\vec{X}), q_1(\vec{X}_1), \ldots, q_{i-1}(\vec{X}_{i-1})$$
   is in *MWFR(P)*.

5. If the query is of the form $? - p(\vec{X})$ then the fact
$$m\_p(\vec{X})$$
is in *MWFR(P)*.

6. Nothing else is in *MWFR(P)*. $\Box$

It is not difficult to see how this transformation can be extended to a compound query $? - Q(\vec{X})$; simply add the (negation replacement of the) rule

$$answer(\vec{X}) \leftarrow Q(\vec{X})$$

to *WFR(P)* before the transformation, and treat $answer(\vec{X})$ as the query.

Example 5.4.3: Let P be the program from Example 5.3.3, and suppose we ask the query $? - p(18)$. Then assuming b and e are EDB predicates, *MWFR(P)* is

$$p(X) \leftarrow m\_p(X), e(X,Y,Z), \Box p(Z), p(Y) \tag{1}$$
$$p(X) \leftarrow m\_p(X), b(X) \tag{2}$$
$$d(p(X), p(Z)) \leftarrow m\_p(X), e(X,Y,Z) \tag{3}$$
$$d(p(X), Q) \leftarrow m\_p(X), e(X,Y,Z), \Box p(Z), dd(p(Y), Q) \tag{4}$$
$$d'(Q,R) \leftarrow dd(Q,R), R \tag{5}$$
$$d'(Q,R) \leftarrow dd(Q,R), \Box R \tag{6}$$
$$m\_p(Z) \leftarrow m\_p(X), e(X,Y,Z) \tag{7}$$
$$m\_p(Y) \leftarrow m\_p(X), e(X,Y,Z), \Box p(Z) \tag{8}$$
$$m\_p(18) \tag{9}$$

together with the original e and b facts. Remember that, in *MWFR(P)*, $\Box Q(\vec{X})$ includes the conjunct $\lhd m\_Q(\vec{X})$.

Consider how a bottom-up evaluation would proceed. The ninth rule will put 18 in m_p, and the seventh rule will add to m_p all the proper factors of 18. The eighth rule can only generate further proper factors of 18, and so m_p is fully evaluated at the first intermediate fixpoint for this program. The iteration proceeds as described in Example 5.4.2, except that the extra m_p predicates in the body restrict the p tuples generated to have arguments that are proper factors of 18. (The d and d' tuples are similarly restricted.) The answer will contain $p(18)$, from which we conclude that 18 is the product of an odd number of primes. $\Box$

By using the magic-set rewriting we avoid having to compute the whole well-founded model. We could demonstrate the correctness of the bottom-up computation of *MWFR(P)* using similar techniques to those of Section 5.4.2. However, we shall defer the questions of correcntess and efficiency until the next section, where we present an improved version of the magic-set rewriting.

5.4.4 Magic Sets: Mark II

The magic set method presented in the previous section is not quite the best possible one, in the sense that the *depends* relations can be unnecessarily large. Also, effort can be saved by using supplementary relations.

In order to reduce redundant computation, common subexpressions may be factored out of the rewriting of the program by using supplementary predicates. For example, the subexpression

$$m\_p(X), e(X, Y, Z)$$

appears in rules 1, 3, 4, 7 and 8 in Example 5.4.3. Rather than compute this join five times, we could compute it once using a rule

$$sup(X, Y, Z) \leftarrow m\_p(X), e(X, Y, Z)$$

and replace the occurrences of the original conjunction by $sup(X, Y, Z)$.

In this section we introduce a variant of the method presented previously that uses supplementary predicates. It also uses a new construct that we call *iterate*. This will allow us more control in the order of rule evaluation. We will iterate the rules generating positive information first, followed by those generating the negative information. This process will itself be iterated until a fixpoint is reached.

In the definition below we will use several meta-predicates:

- It will be convenient to have a meta-predicate "*magic*" rather than a separate predicate m_p for each IDB predicate p in the program. *magic* has two arguments: the first is the atom concerned, and the second is an indicator as to whether the binding appeared due to a positive or negative occurrence of the predicate in a rule. The symbol "+" will denote a positive occurrence, while "−" will denote a negative occurrence. Since negative subgoals will be ground when they are reached, all tuples $magic(P, -)$ will be ground, while tuples of the form $magic(P, +)$ may contain variables.

- We will have two types of dependence predicates:

  - $dp(P, Q)$, which means that P depends positively on Q.

– $dn(P, Q)$, which means that P depends negatively on Q.

dn has a complementary version dn'; $dn'(Q)$ will indicate that all tuples of the form $dn(P, Q)$ are no longer useful, since the status of Q is known. Note than dn' is now a unary predicate rather than a binary predicate. All arguments of tuples of dn and dn' will be ground. The first argument of dp tuples will be ground, while the second may contain variables.

- We no longer need the operator $\lhd$, due to our additional control over the order of rule evaluation. We will need a new, simpler form of negation "$\sim$". For ground atoms P, $\sim P$ holds precisely when P is not *currently* true. Because this negation looks only at the present tuples and does not need to recursively evaluate its argument, it is much easier to implement than well-founded negation.
- $\Box$ is no longer an abbreviation for an expression involving $\lhd$. One may consider $\Box$ as a unary meta-predicate such that $\Box p(\vec{X})$ (eventually) holds when $\neg p(\vec{X})$ is true with respect to the well-founded semantics of the original program; the intuition behind $\Box$ is the same. The argument of $\Box$ will always be a ground atom.

Instances of the meta-predicates described above may be thought of as atoms in the logic called HiLog [25].

If rule r_j of the program P has k subgoals, then we create $k+1$ new supplementary predicates $sup_{j.i}$, for $i = 0, \ldots, k$. $sup_{j.i}$ will have one argument position for each variable that appears either in the rule head, or in both the ith or subsequent subgoals and a subgoal to the left of the ith subgoal.

We now define the supplementary magic rewriting of a program. We base our presentation on [113].

Definition 5.4.4: (Supplementary magic rewriting) Let P be a program. The supplementary magic rewriting of P, denoted *SMR(P)*, contains three sets of rules, denoted P_0, P_1 and P_2. The order of rules within each P_i is unimportant, although we shall present them in an order for which the correspondence with the QSQR/SLS procedure of Section 5.3.2 is clearer. The numbering scheme below is intended to facilitate this correspondence.

The following is included in P_0:

0. If the initial query is ?-$p(\alpha_1, \ldots, \alpha_n)$, where each α_i may be either a constant or a variable, and p is an IDB predicate, then include the rule

$$magic(p(\alpha_1, \ldots, \alpha_n), +).$$

If the initial query is ?-$\neg p(\alpha_1, \ldots, \alpha_n)$, where each α_i must be a constant, and p is an IDB predicate then include the rule

$$magic(p(\alpha_1, \ldots, \alpha_n), -).$$

The following are included in P_1:

1. If the head of r_j is $p(\vec{X})$, and p is an IDB predicate, then include the rule

$$sup_{j.0}(\vec{Y}) \leftarrow magic(p(\vec{X}), \_).$$

Note that the second argument of *magic* is an underscore (_), and not a minus sign ($-$).

2. Suppose rule r_j has k subgoals. Let $q(\vec{Z})$ be the head of r_j. Include the rule

$$q(\vec{Z}) \leftarrow sup_{j.k}(\vec{Y}).$$

3. Let p be an IDB predicate. Suppose $p(\vec{X})$ appears as the ith subgoal of rule r_j. If the arguments of the supplementary predicate $sup_{j.i-1}$ are $\vec{Y}$, then include the rule

$$magic(p(\vec{X}), +) \leftarrow sup_{j.i-1}(\vec{Y}).$$

Suppose that $\neg p(\vec{X})$ appears as the ith subgoal of rule r_j. If the arguments of the supplementary predicate $sup_{j.i-1}$ are $\vec{Y}$, then include the rule

$$magic(p(\vec{X}), -) \leftarrow sup_{j.i-1}(\vec{Y}).$$

4. Suppose rule r_j has k subgoals. If the ith subgoal is positive, say $p(\vec{X})$, and $i \leq k$, then we include the rule

$$sup_{j.i}(\vec{Z}) \leftarrow sup_{j.i-1}(\vec{Y}), p(\vec{X}).$$

If the ith subgoal is negative, say $\neg p(\vec{X})$, and $i < k$, then we include the rule

$$sup_{j.i}(\vec{Z}) \leftarrow sup_{j.i-1}(\vec{Y}), \Box p(\vec{X})$$

if p is an IDB predicate, and if p is and EDB predicate then we include the rule

$$sup_{j.i}(\vec{Z}) \leftarrow sup_{j.i-1}(\vec{Y}), \sim p(\vec{X}).$$

5a. For each positive subgoal $p(\vec{X})$ in r_j, such that p is an IDB predicate, include the rules

$$dp(q(\vec{Y}), p(\vec{X})) \leftarrow magic(q(\vec{Y}), -), sup_{j.i-1}(\vec{Z})$$
$$dp(P, p(\vec{X})) \leftarrow dp(P, q(\vec{Y})), sup_{j.i-1}(\vec{Z})$$

where $q(\vec{Y})$ is the head of r_j and $p(\vec{X})$ is the ith subgoal in r_j.

5b. For each negative subgoal $\neg p(\vec{X})$ in r_j, such that p is an IDB predicate, include the rules

$$dn(q(\vec{Y}), p(\vec{X})) \leftarrow magic(q(\vec{Y}), -), sup_{j.i-1}(\vec{Z})$$
$$dn(P, p(\vec{X})) \leftarrow dp(P, q(\vec{Y})), sup_{j.i-1}(\vec{Z})$$

where $q(\vec{Y})$ is the head of r_j and $\neg p(\vec{X})$ is the ith subgoal in r_j.

5c. Include the rules

$$dn'(Q) \leftarrow magic(Q, -), Q$$
$$dn'(Q) \leftarrow magic(Q, -), \Box Q$$

The following rule is included in P_2:

6. $\Box P \leftarrow magic(P, -), \forall Q(dn(P, Q) \Rightarrow dn'(Q)), \sim P.$

$\Box$

If one is uncomfortable with variables as atoms, then the rule from item 5c in P_1 and the rule in P_2 may be replaced by a sequence of rules in which Q (respectively, P) is instantiated to (most general versions of) all atoms invloving IDB predicates that appear negatively in the program.

Algorithm 5.4.1: (Evaluation of *SMR(P)*.) The rules from Definition 5.4.4 above are evaluated according to the following procedure:

P_0;
`iterate {` `iterate{` P_1 `};`
P_2; `}`

Iteration is performed until there are no additional tuples generated. □

One may think of iterating P_1 as corresponding to the partial construction of the SLP-tree (and recursive SLP-trees) for the given query. P_2 infers negative facts $\neg P$ once no dependences of P upon anything else remain and P has not itself been derived. Note that P_2 must be evaluated after P_1 has reached a fixpoint; otherwise the absence of a negative dependence may cause a negative fact to be inferred even though the negative dependence would be derived in a subsequent step.

dp represents the path relation between the root of each recursively called SLP-tree, and all (leftmost) atoms in nodes of its tree. *dn* represents the path relation between the root and all (leftmost) negative literals in the tree.

Example 5.4.4: We rewrite the program of Example 5.3.3 using the supplementary magic rewriting as follows. We omit those rules that are the same for every program.
P_0 contains the rule

$$magic(p(18), +)$$

P_1 is given in Figure 5.4. The evaluation of this program would proceed in a similar fashion to that in Example 5.4.3, except that the depends relations are smaller, and the supplementary predicates prevent common subexpressions yielding repeated computation. □

The iteration of P_1 may be performed by standard semi-naive evaluation. On the first entry into P_1, the "new" tuples are simply the magic facts from P_0. On subsequent entry to P_1, we also consider negative facts just generated in P_2 as "new." The only non-standard constructs in P_1 are the meta-predicates. The meta-predicates are no problem for semi-naive evaluation; their arguments may be thought of as containing function symbols.

We cannot rewrite P_2 as the sequence

$$temp(P) \leftarrow dn(P, Q), \sim dn'(Q)$$
$$\Box P \leftarrow magic(P, -), \sim temp(P), \sim P$$

because *temp* needs to shrink as dn' grows.

By using appropriate data structures, the relation dn' can be stored as a "marking" of dn. dn' intuitively represents those tuples of dn that are no longer applicable, and can be thought of as "deleted."

In order to know when to fire the rule in P_2, the data structure for dn should maintain a count of the number of unmarked tuples with each first argument. When this reaches zero by the

$$
\begin{aligned}
&sup_{1.0}(X) \leftarrow magic(p(X), \_) \\
&sup_{2.0}(X) \leftarrow magic(p(X), \_) \\
\\
&p(X) \leftarrow sup_{1.1}(X) \\
&p(X) \leftarrow sup_{2.3}(X) \\
\\
&magic(p(Z), -) \leftarrow sup_{2.1}(X, Y, Z) \\
&magic(p(Y), +) \leftarrow sup_{2.2}(X, Y) \\
\\
&sup_{1.1}(X) \leftarrow sup_{1.0}(X), b(X) \\
&sup_{2.1}(X, Y, Z) \leftarrow sup_{2.0}(X), e(X, Y, Z) \\
&sup_{2.2}(X, Y) \leftarrow sup_{2.1}(X, Y, Z), \Box p(Z) \\
&sup_{2.3}(X) \leftarrow sup_{2.2}(X, Y), p(Y) \\
&dp(p(X), p(Y)) \leftarrow magic(p(X), -), sup_{2.2}(X, Y) \\
&dp(P, p(Y)) \leftarrow dp(P, p(X)), sup_{2.2}(X, Y) \\
\\
&dn(p(X), p(Z)) \leftarrow magic(p(X), -), sup_{2.1}(X, Y, Z) \\
&dn(P, p(Z)) \leftarrow dp(P, p(X)), sup_{2.1}(X, Y, Z)
\end{aligned}
$$

Figure 5.4: P_1 generated in Example 5.4.4.

insertion of a dn' tuple, the second subgoal of the rule will become true of the corresponding P. Using this data structure will allow us to avoid recomputing the implication inside the universal quantifier every time P_2 is executed. Note that P_2 converges in one step since it is not recursive.

As noted in [113], we can further optimize the rewriting in several ways. For example, since the there is only one rule for each $sup_{j.0}$, and each such rule has only one subgoal, we could substitute the body of the rule directly into every place that $sup_{j.0}$ is used. This substitution saves j extra temporary relations. A similar observation can be made about $sup_{j.k}$ where rule r_j has k subgoals. While such optimizations would certainly be used in practice, they are not necessary for the efficiency claims made in the next section.

Since using nonground tuples requires additional machinery to perform duplicate elimination and rule firing, it would be desirable to specialize the rewriting above so that nonground tuples do not arise if we can manage without them. Indeed, we can perform such a specialization if we make the assumption that all predicates have a unique binding pattern for the given query. To achieve such a program, occurrences of the same prediacte should be renamed apart if they have different binding patterns; also, subgoal rectification is necessary for subgoals with repeated variables. See [113] for details of how renaming and subgoal rectification are done.

With our assumption about unique binding patterns, we can specialize the *magic*, *dp* and *dn* meta predicates so that their arguments contain only the bound variables from the original rules. For example, if $p(X, Y)$ had binding pattern bf, so that X was bound but Y was free, then *magic*, *dp* and *dn* tuples would look like $magic(p(x), +)$, $dp(p(x), p(x'))$ or $dn(p(x), p(x'))$, where x and x' are possible values for X. With the additional restriction that supplementary predicates $sup_{j.i}$ not contain arguments corresponding to free variables in the head unless those variables have appeared to the left of the ith subgoal in the body, the resulting rewriting method generates a range-restricted program. Hence nonground tuples will not arise in the evaluation of such a program.

The penalty for ensuring no nonground tuples is that some of the rule reordering, subgoal rectification and predicate renaming may have to be done at query-time, when the binding pattern of the query becomes known. This overhead will not be significant if most queries use one of a limited set of binding patterns. In general, whether this overhead is sufficiently compensated for by not having to deal with nonground tuples is a matter for experimental investigation.

The order of the rules in P_1 is of little importance as far as correctness is concerned, and we have chosen an order that makes the comparison with the QSQR/SLS procedure simpler. In practice, rule order can affect the number of rounds before the iteration converges. Such issues are discussed in [90].

We could demonstrate the correctness of Algorithm 5.4.1 by comparison with the methods of the previous section. However we shall show a closer connection between Algorithm 5.4.1 and Algorithm 5.3.1 that identifies a one-to-one correspondence between steps of each method. In addition to demonstrating the correctness of Algorithm 5.4.1, the correspondence also shows that the top-down and bottom-up methods will have the same complexity.

5.5 Comparing Top-Down and Bottom-Up

We now address the efficiency of this magic sets method. We will demonstrate a one-to-one correspondence between steps of Algorithm 5.3.1 and the semi-naive evaluation of Algorithm 5.4.1. We will thus be able to demonstrate that the two algorithms have the same complexity.

Note that our claim is stronger than the more common claim that various methods infer the same sets of tuples. Some authors have considered the issue of "sip-optimality" [88, 105].

Definition 5.5.1: A query-evaluation method is sip-optimal if

- It answers all instances of queries, and
- it generates new queries for every predicate in a rule body whose head matches a query, according to the given "sideways information passing strategy," and
- it infers only the answers and queries necessary by the above items.

□

While sip-optimality is a desirable property, it does not necessarily mean that bottom-up evaluation would be competitive since the same facts may be generated many times by an inefficient method.

We now provide a correspondence between events in the QSQR/SLS procedure and events in the bottom-up evaluation of supplementary magic rewritten program. The correspondence uses the idea of the *stage* of the computation.

Definition 5.5.2: The stage n is a triple (α, β, γ). In the QSQR/SLS procedure, α represents the number of times the outer `repeat` loop has been executed, β represents the number of times the inner `repeat` loop has been executed since the start of the most recent iteration of the outer loop, and γ represents the position of the "program counter" according to the labels in Algorithm 5.3.1.

In the evaluation of Algorithm 5.4.1, α corresponds to the number of outermost iterations, β represents the number of iterations of the inner `iterate` construct since the start of the most recent outermost iteration, and γ represents the position of a "program counter" according to the labels in Definition 5.4.4. Where these labels have subparts (such as 4a, 4b and 4c) the stage (in this case, 4) represents the complete sequence of subparts.

We shall say either algorithm is "at stage n" if it has just completed the statement corresponding to n. When no statements have been executed we denote the stage by ϵ. The *predecessor* of a stage n is the previous stage at which the statement corresponding to n was executed. If $n = (\alpha, \beta, \gamma)$ then the predecessor of n is $(\alpha, \beta - 1, \gamma)$ if $\beta \geq 1$, or $(\alpha - 1, \beta', \gamma)$ if $\beta = 0$ and $\alpha \geq 1$ (for some β'), and ϵ otherwise. □

Theorem 5.5.1: (Correspondence theorem) Let $n = (\alpha, \beta, \gamma)$ be an arbitrary stage,[11] and let m be the predecessor of n. For all predicates p and q, all goals G, and all rules r_j (having s_j subgoals) the following correspondence holds at n:

	QSQR/SLS	Magic – Semi-naive
(a)	The lemma $p(\vec{X})$ is inferred.	$p(\vec{X})$ is inferred.
(b)	The lemma $\neg p(\vec{X})$ is inferred.	$\Box p(\vec{X})$ is inferred.
(c)	A new root $p(\vec{X})$ of positive polarity is constructed.	$magic(p(\vec{X}), +)$ is inferred.
(d)	A new root $p(\vec{X})$ of negative polarity is constructed.	$magic(p(\vec{X}), -)$ is inferred.
(e)	A child goal G' is constructed from the root goal G by resolving with rule r_j yielding an associated substitution θ at G'.	$sup_{j.0}(\vec{Y})\theta$ is inferred, where θ contains substitutions for variables in $\vec{Y}$ only.
(f)	Let G be a non-root goal G with associated rule r_j and leftmost literal L that was originally ith in r_j. A goal G' is constructed as a child of G by resolving L with a lemma L' of the same polarity, yielding an associated substitution θ at G'.	$sup_{j.i}(\vec{Y})\theta$ is inferred, where $1 \leq i \leq s_j$, and θ contains substitutions for variables in $\vec{Y}$ only.
(g)	"p depends positively on q" is inferred.	$dp(p, q)$ is inferred.
(h)	"p depends negatively on q" is inferred.	$dn(p, q)$ is inferred.
(i)	"q is settled" is inferred.	$dn'(q)$ is inferred.

Proof: The proof is by induction on the stage $n = (\alpha, \beta, \gamma)$. We assume that given identical pairs of terms to unify, both procedures produce the same most general unifier. We assume that both procedures rename variables to new variables in the same way, so that we can demonstrate the correspondences above without requiring further renaming of variables. When choosing

[11] Later that ϵ.

most general elements from a set of atoms, we assume that the same choice is made by both QSQR/SLS and magic sets.[12] We also assume that the bottom-up evaluation employs regional duplicate elimination, as described in Section 5.4.1.

Base Case. Suppose the initial query is on the predicate p. At $n = (0,0,0)$ the QSQR/SLS procedure constructs a new root $p(\vec{X})$ of polarity corresponding to that of the query on p. At $n = (0,0,0)$ in the evaluation of *SMR(P)* either $magic(p(\vec{X}),+)$ or $magic(p(\vec{X}),-)$ is inferred, depending on the polarity of p in the query. Part (c) or part (d) of the correspondence will thus hold, and the remainder of the parts above trivially hold at stage $(0,0,0)$.

Induction step. We divide this proof into several parts, corresponding to each step of the method. We assume that the claim above is true for stages before n and prove that the claim is also true at stage n. We shall only consider those items from the correspondence above that change during the particular step under consideration; the remaining correspondences obviously hold by the induction hypothesis.

$\gamma = 1$. Only correspondence (e) is affected. Suppose that $p(\vec{Z})$ is the head of r_j. Note that the variables in $\vec{Y}$ are precisely the variables in $\vec{Z}$.

($\Leftarrow$) Suppose $sup_{j.0}(\vec{Y})\theta$ is inferred at stage n. Then the rewritten rule

$$sup_{j.0}(\vec{Y}) \leftarrow magic(p(\vec{Z}),\_)$$

must have fired due to the insertion of some tuple $magic(p(\vec{W}), sign)$ between m and n, where θ is the restriction of the most general unifier of $\vec{Z}$ and $\vec{W}$ restricted to variables in $\vec{Z}$, and $sign$ is $+$ or $-$. By the induction hypothesis, the root $p(\vec{W})$ of polarity equal to $sign$ is constructed between m and n. Hence at stage n the root $p(\vec{W})$ is resolved with r_j using most general unifier ϕ to yield a child goal G'. θ is the restriction of ϕ to variables in $\vec{Z}$, and is thus the associated substitution at G'. Since $sup_{j.0}(\vec{Y})\theta$ was not subsumed by a previous $sup_{j.0}$ tuple, and is most general among those generated at stage n, θ is not subsumed by any substitution in $C_{j,1}$, and is most general among those generated at stage n.

($\Rightarrow$) Suppose that at stage n the root $p(\vec{W})$ of polarity $sign$ is resolved with r_j to yield a child goal G' with associated substitution θ. Then the root $p(\vec{W})$ of polarity equal to $sign$ must have been constructed between m and n. By the induction hypothesis, $magic(p(\vec{W}), sign)$ must have been inferred between m and n. Thus the rewritten rule

$$sup_{j.0}(\vec{Y}) \leftarrow magic(p(\vec{Z}),\_)$$

fires at stage n yielding the tuple $sup_{j.0}(\vec{Y})\theta$, since θ is the restriction of the most general unifier of $\vec{W}$ and $\vec{Z}$ to variables in $\vec{Z}$. Since θ is not subsumed by any substitution in $C_{j,1}$, and is most general among those generated at stage n, $sup_{j.0}(\vec{Y})\theta$ was not subsumed by a previous $sup_{j.0}$ tuple, and is most general among those generated at stage n.

$\gamma = 2$. Only correspondence (a) is affected.

($\Leftarrow$) Suppose $p(\vec{X})$ is inferred at stage n. Then for some j, the rewritten rule

$$p(\vec{Z}) \leftarrow sup_{j.s_j}(\vec{Y})$$

[12] These assumptions are not critical, but they make the proofs of our results slightly easier as we do not have to worry about renaming variables.

fires due to the insertion of some tuple $sup_{j.s_j}(\vec{Y})\phi$ between m and n, where ϕ contains substitutions for variables in $\vec{Y}$ only. Suppose that $p(\vec{Z})$ is the head of r_j. Then $\vec{Z}\phi = \vec{X}$, no tuple at stage m is more general than $p(\vec{X})$, and $p(\vec{X})$ is a most general representative of the tuples generated at stage n. Hence there is a root node G with atom $p(\vec{V})$ having an empty descendent G' constructed between m and n such that ϕ is the substitution associated with G', r_j is the associated rule of G', by the induction hypothesis. Hence the lemma $p(\vec{Z})\phi = p(\vec{X})$ is inferred. Also, $p(\vec{X})$ is not an instance of a lemma at stage m, and $p(\vec{X})$ is a most general representative of the lemmas inferred at stage n.

($\Rightarrow$) Suppose that the lemma $p(\vec{X})$ is inferred at stage n. Then there is a root node G with atom $p(\vec{V})$ having an empty descendent G' constructed between m and n. Further, if r_j is the associated rule at G', having head $p(\vec{Z})$, and ϕ is the substitution associated with G', then $\vec{Z}\phi = \vec{X}$, $p(\vec{X})$ is not an instance of a lemma at stage m, and $p(\vec{X})$ is a most general representative of the lemmas inferred at stage n. By the induction hypothesis, $sup_{j.s_j}(\vec{Y})\phi$ is inferred between stages m and n. Hence the rewritten rule

$$p(\vec{Z}) \leftarrow sup_{j.s_j}(\vec{Y})$$

fires, giving the tuple $p(\vec{Z})\phi = p(\vec{X})$ at stage n. Also, $p(\vec{X})$ is not an instance of a tuple at stage m, and $p(\vec{X})$ is a most general representative of the tuples inferred at stage n.

$\gamma = 3$. Correspondences (c) and (d) are affected.

($\Leftarrow$) Assume that $magic(p(\vec{X}), sign)$ is inferred at stage n, where $sign$ is either $+$ or $-$. Then for some rule r_j the rewritten rule

$$magic(p(\vec{Z}), sign) \leftarrow sup_{j.i-1}(\vec{Y}).$$

fired due to the insertion of a new tuple $sup_{j.i-1}(\vec{Y})\theta$ between m and n, where $p(\vec{Z})$ (appearing negatively if $sign$ is $-$) is the ith subgoal of r_j and $\vec{Z}\theta = \vec{X}$. Also $magic(p(\vec{X}), sign)$ is not subsumed by any previous tuple, and is a most general representative of the $magic$ tuples inferred at stage n. By the induction hypothesis, some non-root goal node is constructed between m and n with leftmost literal $p(\vec{Z})\theta = p(\vec{X})$ (appearing negatively if $sign$ is $-$). Since there are no previous $magic$ tuples more general than $magic(p(\vec{X}), sign)$, there are no previous root nodes of the same polarity that subsume $p(\vec{X})$. Further, $p(\vec{X})$ is most general among those nodes not subsumed by previous roots of polarity $sign$, and so a new root $p(\vec{X})$ of polarity equal to $sign$ is constructed at stage n.

($\Rightarrow$) Suppose that a new root $p(\vec{X})$ of polarity equal to $sign$ is constructed at stage n. Then there are no previous root nodes of the same polarity that subsume $p(\vec{X})$. Further, $p(\vec{X})$ is most general among those nodes of polarity $sign$ not subsumed by previous roots. Therefore some non-root goal node G must have been constructed between m and n with leftmost literal $p(\vec{X})$ (appearing negatively if $sign$ is $-$). Let θ be the associated substitution at G, let r_j be the associated rule at G, and suppose that $p(\vec{Z})$ was the ith subgoal of r_j, so that $p(\vec{Z})\theta = p(\vec{X})$. By the induction hypothesis, a new tuple $sup_{j.i-1}(\vec{Y})\theta$ was inferred between m and n. Hence, at stage n the rewritten rule

$$magic(p(\vec{Z}), sign) \leftarrow sup_{j.i-1}(\vec{Y}).$$

fires, yielding the tuple $magic(p(\vec{Z}), sign)\theta = magic(p(\vec{X}), sign)$. Since $p(\vec{X})$ is not subsumed by any previous root of the same polarity, and is most general among the new tuples of polarity *sign*, $magic(p(\vec{X}), sign)$ is not subsumed by any previous tuple, and is a most general representative of the *magic* tuples inferred at stage n.

$\gamma = 4$. Only correspondence (f) is affected. Let $S(\vec{X})$ be the ith subgoal of r_j, where $S(\vec{X})$ is of the form either $p(\vec{X})$ or $\neg p(\vec{X})$. Let $S'(\vec{X})$ be the "rewritten version" of $S(\vec{X})$, namely $p(\vec{X})$ if $S(\vec{X}) = p(\vec{X})$, $\Box p(\vec{X})$ if $S(\vec{X}) = \neg p(\vec{X})$ and p is an IDB predicate, and $\sim p(\vec{X})$ if $S(\vec{X}) = \neg p(\vec{X})$ and p is an EDB predicate.

($\Leftarrow$) Suppose $sup_{j.i}(\vec{Y})\theta$ is inferred at stage n, where $1 \leq i \leq s_j$. Then $sup_{j.i}(\vec{Y})\theta$ must have been generated when the rewritten rule

$$sup_{j.i}(\vec{Z}) \leftarrow sup_{j.i-1}(\vec{Y}), S'(\vec{X})$$

fired due to the insertion of a new $sup_{j.i-1}$ or new S' tuple (or both) between m and n such that the join in the rule above is successful.

Case 1: Suppose that there is a new tuple $sup_{j.i-1}(\vec{Y})\phi$ inserted between m and n that participates in the join above. By the induction hypothesis there is a goal node G' with associated substitution ϕ whose leftmost literal is $S(\vec{X})\phi$. Since the join is successful, there must be an instance of $S'(\vec{X})\phi$, say $S'(\vec{X})\phi\psi$ that has been inferred (or in the case of a negated EDB predicate, that is known to be true). By the induction hypothesis, $S(\vec{X})\phi\psi$ is present as a lemma, and hence will be resolved with G' at stage n to give a new goal node G'' with associated substitution being the restriction of $\phi\psi$ to G''.

Case 2: Suppose that there is a tuple $sup_{j.i-1}(\vec{Y})\phi$, and that between stages m and n a new tuple, say $S'(\vec{X})\phi\psi$ is inferred and participates in the join above. (Since this S' tuple is newly inferred, it cannot involve an EDB predicate.) By the induction hypothesis, there is some goal node G' with associated substitution ϕ whose leftmost literal is $S(\vec{X})\phi$. Also by the induction hypothesis $S(\vec{X})\phi\psi$ is inferred as a lemma between m and n, and hence will be resolved with G' to give a new goal node G'' with associated substitution being the restriction of $\phi\psi$ to G''.

In both cases we need to argue that $\theta = \phi\psi$ restricted to G''. By the definition of join, $\theta = \phi\psi$ restricted to the variables in $\vec{Y}$. By the definition of restriction (Definition 5.3.8), the variables $\vec{Y}$ of the supplementary predicate are exactly the variables that $\phi\psi$ are restricted to, and so $\theta = \phi\psi$ restricted to G''.

Finally, since $sup_{j.i}(\vec{Y})\theta$ is not subsumed by any previous tuple, and is most general among those inferred at stage n, it follows that θ is not subsumed by any substitution associated with a node previously in $C_{j,i}$, and that it is most general among those generated at stage n.

($\Rightarrow$) Suppose that the goal G' is constructed from G at stage n by resolving the leftmost literal $S(\vec{X})\phi$ of G with a lemma L' of the same polarity, with most general unifier ψ yielding an associated substitution θ at G'. G has associated substitution ϕ, and $\theta = \phi\psi$ restricted to G'. Then either G was constructed between m and n, or L' was established as a lemma between m and n, or both.

Case 1: G was constructed between m and n. By the induction hypothesis, $sup_{j.i-1}(\vec{Y})\phi$ is inferred between m and n. Also by the induction hypothesis, there is a (rewritten) tuple $S'(\vec{X})\phi\psi$ corresponding to the lemma L'.

Case 2: L' was established between m and n. By the induction hypothesis, there is a tuple $sup_{j.i-1}(\vec{Y})\phi$ corresponding to the goal G. Then the (rewritten) tuple $S'(\vec{X})\phi\psi$ corresponding to L' must have been constructed between m and n, by the induction hypothesis.

In either case, the rewritten rule

$$sup_{j.i}(\vec{Z}) \leftarrow sup_{j.i-1}(\vec{Y}), S'(\vec{X})$$

fires at stage n of the semi naive evaluation, to yield $sup_{j.i}(\vec{Z})\phi\psi$, since the most general unifier of $S\phi$ and S' is ψ. By the definition of restriction, the restriction of $\phi\psi$ to the variables in $\vec{Z}$ is θ, so the tuple $sup_{j.i}(\vec{Z})\theta$ is generated at stage n. Finally, since θ was not subsumed by any previous member of $C_{j,i}$, and is most general among the new additions to $C_{j,i}$, $sup_{j.i}(\vec{Z})\theta$ is not be subsumed by another $sup_{j.i}$ tuple.

$\gamma = 5$. Correspondences (g), (h) and (i) are affected. With an appropriate mechanism for evaluating new "depends positively," "depends negatively" and "settled" relationships, as per the comments after Algorithm 5.3.1, these correspondences follow by similar arguments to the case $\gamma = 4$ above.

$\gamma = 6$. Correspondence (b) is affected. Let $n' = (\alpha, \beta, 5)$.

($\Leftarrow$) Suppose that the rule

$$\Box P \leftarrow magic(P, -), \forall Q(dn(P,Q) \Rightarrow dn'(Q)), \sim P$$

fires at stage n, yielding a tuple $\Box P$ that was not inferred previously. Then at stage n', $magic(P, -)$ holds, P does not hold, and $dn'(Q)$ holds for every Q for which $dn(P,Q)$ holds. By the induction hypothesis, there is a root P of negative polarity at n'. Since the tuple P is not inferred by n', it follows by the induction hypothesis that there is no lemma for P at n'. Further, since $\forall Q(dn(P,Q) \Rightarrow dn'(Q))$ holds at n', by the induction hypothesis it follows that for every Q that P depends negatively upon, Q is settled. Hence $\neg P$ will be inserted as a lemma at stage n, having not been inserted previously.

($\Rightarrow$) Suppose that $\neg P$ is added as a lemma at stage n, but was not a lemma before n. Then at stage n', there was a root of negative polarity for P such that there is no lemma for P, and every ground atom Q that P depended negatively upon is settled. By the induction hypothesis, $magic(P, -)$, $\forall Q(dn(P,Q) \Rightarrow dn'(Q))$ and $\sim P$ hold at n', and thus the rule

$$\Box P \leftarrow magic(P, -), \forall Q(dn(P,Q) \Rightarrow dn'(Q)), \sim P$$

fires at n. Since $\neg P$ had not been inserted as a lemma previously, $\Box P$ is inferred for the first time at stage n. ∎

Corollary 5.5.2: Algorithm 5.4.1 is correct with respect to the well-founded semantics for range-restricted nonfloundering programs having finite negation trees.
Proof: By Theorem 5.5.1 and Theorem 5.3.3. ∎

In particular, Corollary 5.5.2 holds for all permissible programs.

Corollary 5.5.3: The semi-naive bottom-up computation of *SMR(P)* has the same time complexity as QSQR/SLS. ▮

Seki [105] demonstrates a similar correspondence to that of Theorem 5.5.1 in the context of (not necessarily range-restricted) programs without negation. Ignoring our extension to programs with negation, our procedures are slightly different from Seki's: we use regional duplicate elimination while Seki uses incremental duplicate elimination. Apart from generating fewer duplicate tuples, using regional evaluation has technical advantages for the proof of Theorem 5.5.1, as we do not need to concern ourselves with the order in which tuples are added. Also, Seki does not distinguish between IDB and EDB predicates. Both Algorithm 5.3.1 and Algorithm 5.4.1 work for programs that are not range restricted, as long as the program has finite negation trees and does not flounder, and Theorem 5.5.1 holds in this case.

5.6 Generalizations

Much of the material presented here is not specific to negation. There are other operators, such as set-grouping and aggregation, that have traditionally been stratified in order to prevent semantic difficulties. The idea of modular stratification can be extended to these operators too. For example, suppose we have a relation $part(X, Y, N)$ that is true when X has N copies of Y as an immediate subpart.[13] The "parts-explosion" problem is to determine, for an arbitrary pair of parts x and y, how many y's appear in x. For example, if a bicycle has two wheels, and each wheel has forty-seven spokes, then we would like to infer that a bicycle has ninety-four spokes. Assuming we have access to some standard arithmetical operations, we can solve the parts-explosion problem using the following program.

$$\begin{array}{l} in(X, Y, null, N) \leftarrow part(X, Y, N) \\ in(X, Y, Z, N) \leftarrow part(X, Z, P), contains(Z, Y, M), N = P * M \\ contains(X, Y, N) \leftarrow N = \texttt{sum}\ P : in(X, Y, \_, P) \end{array}$$

(The sum in the third rule is grouped by X and Y; for each X and Y we sum all corresponding P.) The `sum` operation here is not stratified. *contains* depends on itself through aggregation, via the predicate *in*. However, assuming *part* is acyclic in its first two arguments, the summation operates on successively lower arguments (i.e., smaller subparts), and so there is no looping through summation. This is the aggregate analog of modular stratification.

Informally, one can argue that the parts explosion problem cannot be expressed naturally[14] with only stratified aggregation. Recall the game program from Example 5.2.1, which was not expressible by a stratified program because an *unbounded* number of recursions through negation is required. The parts explosion problem cannot be expressed naturally by a program stratified with respect to summation as we may need an unbounded number of summations, depending upon the depth of the part tree.

The methods presented in this chapter generalize to the class of programs with function symbols. However, if function symbols are recursively applied, then termination cannot be guaranteed.

[13] Again, we adopt the convention that we are only interested in smaller, simpler subparts.

[14] We ignore here the possibility of using the integers, multiplication and summation to encode Turing machines.

With a slight change of syntax, the rewritten program of Algorithm 5.4.1 may be interpreted as statements in the database programming language GLUE [79, 80]. In fact, Algorithm 5.4.1 is the basis of a compiler that has been implemented by the author and others as part of the NAIL! system at Stanford University.

Chapter 6

Tail recursion elimination

6.1 Introduction

Right-linear evaluation is a specialized optimization technique for evaluating queries on right-linear programs. Right-linearity is a syntactic property of a program that corresponds to a type of "linear tail recursion" (in the terminology of programming languages). Right-linear evaluation can significantly speed up query-evaluation compared to a more general method such as magic sets. On the other hand, such optimization has been available only for right-linear programs.

Apart from the utility of an optimization itself, there is another important issue, namely how to mix different optimization strategies within the one program. For example, it may be possible to evaluate a portion of a program most efficiently using right-linear evaluation, while the rest of the program may not be right-linear; we would still like to be able to use, say, magic sets on the remainder of the program.

In this chapter we propose an optimization technique that integrates magic sets and a form of tail-recursion elimination significantly more general than right-linear evaluation. By not representing intermediate results of a bottom-up computation, substantial savings over magic sets alone can be realized.

We consider cases where top-down without memoing beats magic set bottom-up methods because "intermediate" predicates are not fully computed. In some cases, top-down Prolog-style evaluation effectively applies tail-recursion elimination. It is this tail-recursion elimination that is exploited for right-linear programs in [76, 77]. We use this observation about top-down evaluation to motivate a bottom-up formalization of tail-recursion elimination. This formalization improves on top-down evaluation in that it remains efficient even when top-down evaluation does not terminate.

We describe a method by which magic-templates can be augmented with such a mechanism to take advantage of tail recursion. We define the class of "nonrepeating" programs and show that for nonrepeating programs the augmented version of magic templates does no worse than ordinary magic templates, while it often does much better. We define a sufficient syntactic condition for nonrepetiton, which we call "weak right linearity." The class of weakly right linear programs properly includes the class of right-linear programs.

6.2 Top-Down Can Beat Bottom-Up

You may be startled by the title of this section, especially if you have seen the paper "Bottom-up Beats Top-down for Datalog," [112]. However, there is no inconsistency here: the model of top-down computation in [112] uses a (weak) form of memoing, whereas the one used in this section does not.

There are cases where top-down without memoing can "beat" bottom-up with magic sets (with or without duplicate elimination) for the following reason: Top down returns the binding relation at the leaves of derivation trees directly, while magic sets computes all the intermediate relations. Consider the following example:

Example 6.2.1: Let P be

$$\begin{array}{l} p(X,Z) \leftarrow e(X,Y), p(Y,Z) \\ p(n,X) \leftarrow t(X) \\ e(1,2) \\ \vdots \\ e(n-1,n) \\ t(1) \\ \vdots \\ t(m) \end{array}$$

A top-down method such as SLD-resolution[1] would construct the following derivation tree for the query ?-$p(1,X)$.

$$\begin{array}{ccc} & p(1,X) & \\ & | & \\ & e(1,Y), p(Y,X) & \\ & | & \\ & p(2,X) & \\ & | & \\ & \vdots & \\ & | & \\ & p(n,X) & \\ & | & \\ & t(X) & \\ / & | & \backslash \\ \{X=1\} & \cdots & \{X=m\} \end{array}$$

To find all the answers the amount of work is $O(m+n)$. By contrast, the magic sets method would compute all the intermediate relations $p(i,X)$ for $i = 2, \ldots n$, each of which is of size m. Hence magic sets computes $\Omega(mn)$ tuples, and is inferior to SLD-resolution on this example. □

The complexity of the top-down evaluation in the above example is dependent both upon the query and upon the e relation.

Example 6.2.2: Consider the program from Example 6.2.1.

[1]For a formal definition of SLD-resolution see Section 6.4.

(a) If the query was ?-$p(X, Y)$ rather than ?-$p(1, X)$ then the SLD-tree would have size of the order of $mn + n^2$ while magic sets would have time complexity $O(mn)$.

(b) If we added the $n - 2$ extra tuples

$$e(1, 3), e(2, 4), \ldots, e(n - 2, n)$$

to the program above, then the SLD-tree for the original query ?-$p(1, X)$ would still be finite, but will have mF_n leaves[2] and hence be of size exponential in n. By contrast, magic sets (with duplicate elimination) would still take $O(mn)$ space and time.

(c) If we added the tuple $e(n, 1)$ we would create a cycle in the e relation. SLD-resolution would not terminate, while magic sets would still take $O(mn)$ space and time. □

Items (b) and (c) from Example 6.2.2 suggest that SLD-resolution has very limited applicability. However, we will be able to formalize the tail-recursion optimization suggested in Example 6.2.1 within a bottom-up framework in such a way that duplicate elimination will remove these drawbacks.

6.3 Bottom-Up with Tail Recursion

We describe in this section a method to incorporate the right-recursive aspect of SLD-resolution in a bottom-up framework to take advantage of the tail recursion present in a program. This transformation will allow the tail recursive evaluation of the right-recursive portion of the program while the remainder of the program is evaluated according to standard magic set techniques. The advantages of bottom-up methods over top-down methods are discussed in [113].

Bry's translation of SLD-resolution [19] explicitly represents the intermediate tuples, and hence does not take advantage of this potential optimization.

We extend the magic templates method of Ramakrishnan [88] in which the magic predicates have the same arity as their corresponding IDB predicates, and nonground tuples are allowed. We use a version of magic templates with supplementary predicates, as in [113]. We also view *magic* as a meta-predicate in the sense of HiLog [25] rather than having a separate magic predicate m_p for each p. We consider only Horn programs (with function symbols) in this chapter. Programs with negation can be handled in a similar way using the methods of Chapter 5.

The first step is to decide which predicates are to be evaluated right-recursively. In Example 6.2.1 we would choose p, for example. Predicates that are not right-recursive will be evaluated as before, using magic templates.

The right-recursive predicates will not have corresponding magic tuples. They shall have corresponding tuples in a HiLog-style binary relation we call *query*. The tuple $query(P, A)$ will mean that we are trying to solve the atom P, and that any answer substitutions found should generate an answer tuple for A, but not necessarily for P.[3]

[2] Here, F_n is the nth Fibonacci number, which for large n is well approximated by $\alpha\phi^n$ where $\phi \approx 1.618$ and $\alpha \approx 0.447$.

[3] Recall that we allow non-ground tuples, so we could have tuples like $query(p(a, X, Y), q(X))$.

In the following rules, b_i will denote a (possibly empty) conjunction of atoms, each p_i will denote a predicate, and a superscript of (n) on a subgoal p will mean that p is the nth subgoal in the rule. A vector $\vec{X}$ represents a vector of *terms*, not necessarily lone variables.

We now review the supplementary version of magic templates described in [113]. Our presentation is slightly different in that we make *magic* a meta-predicate rather than having a separate magic predicate m_p for each IDB predicate p.[4] For each rule r_j having s_j subgoals we create s_j new relations named $sup_{j.i}$ for $i = 0, \ldots, s_j - 1$. The arguments of $sup_{j.i}$ are those variables that are either in the head of r_j, or appear both after the ith subgoal and also in the first i subgoals.

For each rule

$$h(\vec{X}) \leftarrow p_1(\vec{X}_1), \ldots, p_n(\vec{X}_n)$$

we generate some rewritten rules. For each IDB predicate p_i we generate the rule

$$magic(p_i(\vec{X}_i)) \leftarrow sup_{j.i-1}(\vec{Y})$$

where $\vec{Y}$ are the arguments of $sup_{j.i-1}$. For $i = 1, \ldots, n-1$ we generate the rule

$$sup_{j.i}(\vec{Y}) \leftarrow sup_{j.i-1}(\vec{Y}'), p_i(\vec{X}_i)$$

where $\vec{Y}$ and $\vec{Y}'$ are the respective arguments of the supplementary predicates. We generate the rule

$$h(\vec{X}) \leftarrow sup_{j.n-1}(\vec{Y}), p_n(\vec{X}_n)$$

and the rule

$$sup_{j.0}(\vec{Y}) \leftarrow magic(h(\vec{X})).$$

For rules that do not contain right-recursive predicates, the magic template rewriting is unchanged. We consider rules containing right-recursive predicates below.

Firstly, the arguments of the supplementary predicates for rules with head predicates that are right recursive will have an extra argument A corresponding to the answer tuple to be generated. Also, for these rules we omit variables that do not appear further to the right in the rule body, even if they occur in the head.

We perform the magic templates rewriting as above, with the modifications noted below. We consider several nonexclusive cases for each rule r_j containing a right recursive predicate.

1. The body of r_j contains one or more right recursive predicates. For each right-recursive p_i that is either not rightmost in r_j, or rightmost in r_j where the head of r_j is not right recursive, we rewrite as follows. Suppose r_j is

$$h(\vec{X}) \leftarrow b_1, p_i^{(m)}(\vec{Y}), b_2$$

Replace the magic templates rule with head predicate *magic* with

$$query(p_i(\vec{Y}), p_i(\vec{Y})) \leftarrow sup_{j.m-1}(\vec{Z})$$

where $\vec{Z}$ are the arguments of the $m-1$st supplementary predicate.

[4] The version of magic templates presented here is a simplification of the version in Section 5.4.4 for programs without negation.

2. The head of r_j contains an atom $p_0(\vec{X})$ whose predicate is right-recursive. The rules for supplementary predicates now use *sup* predicates with an extra argument. Each rule for $sup_{j.i}$ now has the form

$$sup_{j.i}(\vec{Y}, A) \leftarrow sup_{j.i-1}(\vec{Y}', A), p(\vec{X})$$

for $i \geq 1$, where $p(\vec{X})$ is the ith subgoal in r_j. Instead of the magic templates initialization rule for $sup_{j.0}$ use the rule

$$sup_{j.0}(\vec{X}', A) \leftarrow query(p_0(\vec{X}), A).$$

where $\vec{X}'$ and A are the arguments of $sup_{j.0}$.

Also, if the rightmost subgoal of r_j is not right recursive, then replace the magic templates rule for p_0 with

$$A \leftarrow sup_{j.m-1}(\vec{Z}, A), q(\vec{Y})$$

where $q(\vec{Y})$ is the rightmost (and mth) subgoal of r_j.

3. The head predicate of r_j is right-recursive and the rightmost subgoal of the body is right-recursive. Suppose r_j is

$$p_0(\vec{X}) \leftarrow b_0, p_k^{(m)}(\vec{Y})$$

Replace the two magic templates rules with head predicates p_0 and *magic* with the rule

$$query(p_k(\vec{Y}), A) \leftarrow sup_{j.m-1}(\vec{Z}, A).$$

If the query ?-$p_i(\vec{X})$ is on a right recursive predicate then include the fact $query(p_i(\vec{X}), p_i(\vec{X}))$ as a seed fact. If the query ?-$q(\vec{Y})$ is on a non-right-recursive predicate then include the magic seed fact $magic(q(\vec{Y}))$ as for standard magic templates. We call the the transformation above "magic templates with right-recursion." We denote the result of applying the rewriting to a program P by MTRR(P). Note that item 2 generates a HiLog rule [25], not a first-order rule.

Before we discuss the correctness of this transformation, let us see how it handles the program of Example 6.2.1.

Example 6.3.1: We label p as right-recursive. The rewritten version of P from Example 6.2.1 for the query ?-$p(1, Z)$ is

$$\begin{array}{l}
sup_{1.0}(X, Z, A) \leftarrow query(p(X, Z), A) \\
sup_{1.1}(Y, Z, A) \leftarrow sup_{1.0}(X, Z, A), e(X, Y) \\
query(p(Y, Z), A) \leftarrow sup_{1.1}(Y, Z, A) \\
sup_{2.0}(X, A) \leftarrow query(p(n, X), A) \\
A \leftarrow sup_{2.0}(X, A), t(X) \\
query(p(1, Z), p(1, Z))
\end{array}$$

together with the EDB facts for e and t. Note that X does not appear as an argument of $sup_{1.1}$, even though it appears in the head of the original rule. (However, X may appear in the atom

A.) The tuples generated are

$$\begin{array}{l} query(p(i,Z),p(1,Z)) \text{ for } i = 1,\ldots,n. \\ sup_{1.0}(i,Z,p(1,Z)) \text{ for } i = 1,\ldots,n. \\ sup_{1.1}(i,Z,p(1,Z)) \text{ for } i = 2,\ldots,n. \\ sup_{2.0}(Z,p(1,Z)) \\ p(1,j) \text{ for } j = 1,\ldots,m. \end{array}$$

Thus we have reduced the cost of semi-naive bottom-up evaluation to $O(n+m)$.

Consider also how the rewritten program performs for the modified query and for the modified e relation of Example 6.2.2. If the query is $p(X,Y)$ as in part (a), then the complexity is $\Omega(mn+n^2)$ as for SLD-resolution, since there will be $O(n^2)$ tuples of the form $query(p(i,Y),p(j,Y))$ generated, for $1 \leq j < i \leq n$. This may be worse than ordinary magic sets, which has time and space complexity $O(mn)$.

On the other hand, if we ask the original query once the additional e tuples are added as in part (b), then as long as duplicate elimination is applied, the complexity is still $O(m+n)$, and not exponential in n like SLD-resolution. In fact, we still get $O(m+n)$ complexity when the e relation has cycles (and has size $O(n)$) as in part (c). □

6.4 Correctness

We shall show that our method is correct by comparison with SLD-resolution of [60]. In the case where there are no right-recursive predicates, our result implies the correctness of magic templates, as shown previously in [88]. Our version of SLD-resolution uses a left-to-right computation rule.

Definition 6.4.1: (SLD-tree) Let P be a program and G a goal. The *SLD-tree* for G with respect to P is defined as follows:

- Each node of the tree is a (possibly empty) goal, together with a computed substitution.
- The root node is G, with empty computed substitution.
- Let $G' = \leftarrow A_1, \cdots, A_n$ $(n \geq 1)$ be a node in the tree. Then G' has a child for each rule r such that the head of r and A_1 are unifiable. Let $H \leftarrow B_1, \cdots, B_m$ be a variant of r using new variables. The child node G'' is

  $$\leftarrow (B_1, \cdots, B_m, A_2, \cdots, A_n)\theta$$

  where θ is the most general unifier of A_1 and H. The computed substitution at G'' is $\phi\theta$ where ϕ is the computed substitution at G'. We say that each $B_i\theta$ in G'' is a *rule-child* of A_1 in G'. For $i > 1$, if C_i in G_i has rule-child A_i in G', then $A_i\theta$ in G'' is also a rule-child of C_i in G_i.
- Empty nodes have no descendants.

We shall assume without loss of generality that the most general unifiers chosen are idempotent. We say C in G_C is a *rule-ancestor* of D in G_D if there is a sequence $C_0, \ldots, C_k$ of atoms, and a sequence of goals $G_{C_0}, \ldots, G_{C_k}$ such that $C_0 = C$, $G_0 = G_C$, $C_k = D$, $G_k = G_D$, and for $i = 1, \ldots, k$, C_{i-1} in G_{i-1} is a rule-child of C_i in G_i. Similarly, we can define *rule-descendant.*

A *subrefutation* for the leftmost atom A in a goal G' is a shortest (downward) path in the SLD-tree from G' to a descendant G'' of G' that contains no rule-descendants of A. We associate with the subrefutation the computed substitution at G''.

SLD-resolution is the process of finding the computed substitutions at all empty nodes of an SLD-tree. □

Note the difference between ancestors in the SLD-tree (a relationship between goals) and rule-ancestors (a relationship between atoms in goals).

We assume that a subset of the IDB predicates of the program have been labelled as "right-recursive." If an atom has a right-recursive predicate, we shall say that the atom is also right-recursive. We make use of the concept of a "right ancestor."

Definition 6.4.2: (Right ancestor). Let a be a selected (i.e., leftmost) atom in a node N of the SLD-tree for an atomic query q. We associate with a a *right ancestor* that is either a itself, or a rule-ancestor of a in N. If a is not right recursive, then the right ancestor of a is a. If a is right recursive, then:

- If a is q, i.e., N is the root of the tree, then the right ancestor of a is a itself.
- If a was introduced into a goal by resolving the rule r with a goal node G, and this occurrence of a is not rightmost in r, then the right ancestor of a is a.
- If a was introduced into a goal by resolving the rule r with a goal node G having selected atom h, but h is not right-recursive, then the right ancestor of a is a.
- If a was introduced into a goal by resolving the rule r with a goal node G having selected atom h that is right-recursive, and this occurrence of a is rightmost in r, then the right ancestor of a is the right ancestor of h in G. □

To simplify the proof below, we shall modify the rewritten program in such a way that the answers it generates are unchanged. First, we unfold all supplementary predicates, substituting the body of the rule for a supplementary predicate for the occurrences of the head supplementary predicate in other rules' bodies. Unfolding in this way does not change any of the values computed for the other predicates, although computing the values without supplementary predicates may require some redundant computation. The unfolding is effective, since supplementary predicates are not (directly) recursive in one another.

We shall also rewrite the *magic* predicates as *query* predicates, substituting *query*(P, P) for every occurrence of *magic*(P) in the head and body of each rule. Again, this does not change the values of any of the other tuples since *query* previously held (and is referenced) only with arguments that have right-recursive predicates.

Once MTRR(P) has been unfolded and had its *magic* atoms replaced, we denote the result by MTRR$'(P)$.

Theorem 6.4.1: Let P be a program and let P_R = MTRR$'(P)$ where S is the set of right recursive predicates. Let ?-Q be a query, where Q is an IDB atom. Then the following correspondence holds between the SLD tree with root $\leftarrow Q$, and the bottom-up evaluation of P_R for the query Q. Let L, M and N be IDB atoms.

	SLD-tree	Evaluation of P_R
(a)	There is a goal G with leftmost atom L, computed substitution θ, the right ancestor of L is M, and $N = M\theta$.	$query(L, N)$ is inferred.
(b)	There is a goal G with leftmost atom L, a subrefutation for L with computed substitution θ, the right ancestor of L is L itself, and $N = L\theta$.	N is inferred.

Proof: In what follows we assume that the rule variant used in the top-down computation uses the same variables as the rewritten rules used in the bottom-up computation. We also assume that top-down and bottom-up methods compute the same most general unifiers. These assumptions are not critical, but they simplify the proof by allowing us to ignore variable renaming.

The proof from left to right is by induction on the depth of the SLD-tree. The proof from right to left is by induction on the number of iterations of the bottom-up evaluation of MTRR$'(P)$.

($\Rightarrow$) The base case is straightforward: At depth 1 the SLD-tree for Q has one node, namely $\leftarrow Q$, with leftmost atom Q, the identity computed substitution and Q itself as right ancestor. In MTRR$'(P)$, $query(Q, Q)$ holds.

Let us now consider the induction step. Suppose that the implication from left to right holds for the SLD-tree up to depth n. We show that the implication also holds at depth $n + 1$.

Let G' be a goal at depth $n + 1$ obtained by resolving the goal G of depth n with a variant of rule r_j. Let G be $\leftarrow D_1, \cdots, D_k$. Let the variant of r_j used be

$$h^{(j)} \leftarrow b_1^{(j)}, \cdots, b_m^{(j)}.$$

Suppose the computed substitution at G is ϕ, the most general unifier of $h^{(j)}$ and the leftmost atom D_1 in G is ψ. The computed substitution at G' is then $\theta = \phi\psi$. Since we used a variant of r_j with new variable names, it follows that $b_1^{(j)}\psi = b_1^{(j)}\theta$ and $h^{(j)}\psi = h^{(j)}\theta$. Let M be the right ancestor of D_1 in G. By the induction hypothesis, $query(D_1, M\phi)$ holds. There are two cases:

$m \geq 1$ (In other words, the body of r_j is nonempty.) We only need to consider correspondence (a) since no new subrefutations are generated. Then the leftmost atom in G' is $b_1^{(j)}\psi = b_1^{(j)}\theta$. The construction of MTRR$'(P)$ implies that if $m > 1$, or if $m = 1$ and $b_1^{(j)}$ is not right recursive, then $query(b_1^{(j)}, b_1^{(j)})\theta$ holds, by the firing of an (unfolded, de-magicked) rewritten rule of the form

$$query(b_1^{(j)}, b_1^{(j)}) \leftarrow query(h^{(j)}, h^{(j)})$$

if h_j is not right-recursive, or if $h^{(j)}$ is right-recursive then by a rule of the form

$$query(b_1^{(j)}, b_1^{(j)}) \leftarrow query(h^{(j)}, A).$$

If $m = 1$ and $b_1^{(j)}$ is right recursive, then the right ancestor of $b_1^{(j)}\theta$ in G' is M. In this case, we also know that $query(b_1^{(j)}, M)\theta$ holds by the firing of the (unfolded, de-magicked) rule of the form

$$query(b_1^{(j)}, A) \leftarrow query(h^{(j)}, A).$$

$m = 0$ (I.e., the body of r_j is empty.) We consider correspondence (b) first. We have a subrefutation for D_1 with computed substitution θ, and possibly subrefutations for other ancestors of G' also with computed substitution θ. We use a second (inner) induction argument to show that for every ancestor G_A of G' with leftmost atom A that is its own right ancestor, $A\theta$ holds. The induction is on the distance from G' on the chain of ancestors of G'.

Base case: $G_A = G'$, i.e., $A = D_1$. If D_1 is its own right ancestor, then $query(D_1, D_1)$ holds, by the induction hypothesis. Then the rewritten rule

$$h^{(j)} \leftarrow query(h^{(j)}, h^{(j)})$$

fires if $h^{(j)}$ is not right-recursive, or if $h^{(j)}$ is right recursive then the rule

$$A \leftarrow query(h^{(j)}, A)$$

fires. In either case, $D_1\psi = D_1\theta$ holds.

Induction step: Suppose G_A is an ancestor of G' with A appearing leftmost in G_A, that there is a subrefutation of A with computed substitution θ through G', and that the (inner) induction hypothesis holds for all descendents of G_A on the path to G'. Suppose the right ancestor of A in G_A is A itself. Let G_B be the closest descendent of A along the path to G' such that B is leftmost in G_B, and B is its own right ancestor in G_B. Then by the (inner) induction hypothesis, $B\theta$ holds. Consider the goal G_C that is resolved with a rule variant r_l to generate (a more general instance of) B. Let C be the leftmost atom in G_C. Since we have a subrefutation for A, the subgoal of r_l matching B must be rightmost in r_l. The right ancestor of C must be A, since the right ancestor relationship is idempotent, and since G_B is the closest descendant of A having leftmost atom that is its own right ancestor. By the (outer) induction hypothesis, $query(C, A\theta_C)$ holds, where θ_C is the computed substitution at G_C, and is more general that θ.

Consider the rewritten rule

$$A \leftarrow query(h^{(l)}, A), b_1^{(l)}, \cdots, b_q^{(l)}$$

where r_l is $h^{(l)} \leftarrow b_1^{(l)}, \cdots, b_q^{(l)}$. None of $b_1^{(l)}, \ldots, b_{q-1}^{(l)}$ are leftmost in r_l. Hence the fact that there are subrefutations for appropriate instances of $b_1^{(l)}$ to $b_{q-1}^{(l)}$ implies that $b_i^{(l)}\theta_i$ holds for $i = 1, \ldots, q-1$ (where θ_i is defined below), by the (outer) induction hypothesis. For $i = 1, \ldots, q-1$,

$$\theta_i = \theta_{i-1}\psi_i$$

for some substitutions ψ_i, and $\theta_0 = \theta_C$. Now $B\theta = b_q^{(l)}\theta$ holds, and since the substitutions on all other atoms in the rule above are more general than θ, it follows that $A\theta$ is inferred.

We now look at correspondence (a). Consider the status of $G' = \leftarrow D_2\psi, \cdots, D_k\psi$. Let r_l now be the rule variant in which (a more general version of) D_2 was first introduced, let H be the corresponding goal in which (a more general version of) D_2 first appears, and let θ_0 be the computed substitution at H. Let H_D be the parent of H that is resolved with r_l, with leftmost atom D and having computed substitution θ_D. By the induction hypothesis, $query(D, M\theta_D)$ holds, where M is the right ancestor of D in H_D. Let $b_1^{(l)}, \cdots, b_q^{(l)}$ be the subgoals to the left of D_2 in r_l, so that D_2 is an instance of $b_{q+1}^{(l)}$, and suppose $h^{(l)}$ is the head of r_l. Then by G' there is a subrefutation for each $b_i^{(l)}$ with computed substitution θ_i such that for $i = 1, \ldots, q$,

$$\theta_i = \theta_{i-1}\psi_i$$

for some substitutions ψ_i, and θ_q is more general than θ. Since none of the subgoals $b_i^{(l)}$ in r_l to the left of D_2 is rightmost in r_l, each $b_i^{(l)}$ is its own right ancestor and $query(b_i^{(l)}, b_i^{(l)})\theta_{i-1}$ holds for $i = 1, \ldots, q$ by the induction hypothesis.

By the induction hypothesis we have subrefutations of depth at most n for each $b_i^{(l)}\theta_{i-1}$ for $i = 1, \ldots, q-1$ with computed answer substitutions $\theta_{i-1}\psi_i$, and hence $b_i^{(l)}\theta_i$ holds. The subrefutation of $b_q^{(l)}$ is at depth $n+1$, and $b_q^{(l)}\theta$ holds by correspondence (b) discussed above. Consider the (unfolded, de-magicked) rewritten rule

$$Q_1 \leftarrow Q_2, b_1^{(l)}, \cdots, b_q^{(l)}$$

where Q_1 and Q_2 are as follows. Q_1 is $query(b_{q+1}^{(l)}, A)$ if $b_{q+1}^{(l)}$ and $h^{(l)}$ are both right recursive and $b_{q+1}^{(l)}$ is rightmost in r_l; otherwise Q_1 is $query(b_{q+1}^{(l)}, b_{q+1}^{(l)})$. If $h^{(l)}$ is right recursive then Q_2 is $query(h^{(l)}, A)$; otherwise Q_2 is $query(h^{(l)}, h^{(l)})$.

This rule fires with Q_2 unifying with $query(D, M\theta_D)$ to give $Q_1\theta$. If $b_{q+1}^{(l)}$ and $h^{(l)}$ are both right recursive and $b_{q+1}^{(l)}$ is rightmost in r_l, then M is the right ancestor of D_2 in G' and $query(b_{q+1}^{(l)}, A)\theta = query(D_2, M\theta)$ is generated. Otherwise, D_2 is its own right ancestor in G' and $query(b_{q+1}^{(l)}, b_{q+1}^{(l)})\theta = query(D_2, D_2)$ is generated.

($\Leftarrow$) The base case is also straightforward: After the first iteration of MTRR$'(P)$ only $query(Q, Q)$ holds. The SLD-tree for Q has root $\leftarrow Q$, with leftmost atom Q, the identity computed substitution and Q itself as right ancestor.

Let us now consider the induction step. Suppose that the implication from right to left holds for iterations up to n. We show that the implication also holds at iteration $n+1$.

We consider correspondence (a) first. Suppose $query(L, N)$ is inferred at stage $n+1$. Then there are two cases:

1. L has a right-recursive predicate, a more general version of L appears rightmost in some rule variant r_j given by

$$h^{(j)} \leftarrow b_1^{(j)}, \cdots, b_m^{(j)}$$

$h^{(j)}$ is also right-recursive, and $query(L, N)$ is generated by firing the rewritten rule

$$query(b_m^{(j)}, A) \leftarrow query(h^{(j)}, A), b_1^{(j)}, \cdots, b_{m-1}^{(j)}.$$

Suppose that the matching tuples used in the body are $query(H, B)$, $b_1, \cdots, b_{m-1}$ respectively. Let θ_0 be the most general unifier of $query(H, B)$ and $query(h^{(j)}, A)$. Let θ_i be θ_{i-1} composed with the most general unifier of $b_i^{(j)}\theta_{i-1}$ and b_i, for $i = 1, \cdots, m-1$, and let $\theta = \theta_{m-1}$. Thus $query(L, N) = query(b_m^{(j)}, B)\theta$. $query(H, B)$ holds at stage n. Hence, by the induction hypothesis there is a goal G with leftmost atom H, computed answer substitution ϕ, the right ancestor of H is C, and $B = C\phi$. G can be resolved with r_j, using most general unifier ψ to get a goal $G_1 = b_1^{(j)}\theta_0, \cdots, b_m^{(j)}\theta_0, \cdots$ where $\theta_0 = \phi\psi$ is the computed substitution at G_1. By the induction hypothesis, since b_1 holds at stage n, $b_1^{(j)}\theta_0$ has a subrefutation from G' with computed substitution θ_1. Consider G_2, the node with computed substitution θ_1 and leftmost atom $b_2^{(j)}\theta_1$, at which the previous subrefutation was generated. Again, by the induction hypothesis, there is a subrefutation for $b_2^{(j)}\theta_1$ with computed substitution θ_2. We proceed in this way until we reach a goal node G_m. The computed substitution at G_m is $\theta_{m-1} = \theta$, and the leftmost atom in G_m is $b_m^{(j)}\theta = L$. The right ancestor of L is C, and $N = C\theta$.

2. L has a predicate that is not right-recursive, or L is right recursive but either (a more general version of) L does not appear rightmost in a rule variant r_j, or the head of r_j is not right-recursive, where r_j given by

$$h^{(j)} \leftarrow b_1^{(j)}, \cdots, b_m^{(j)}$$

and *query*(L, N) is generated by firing the rewritten rule

$$query(b_k^{(j)}, b_k^{(j)}) \leftarrow Q, b_1^{(j)}, \cdots, b_{k-1}^{(j)}$$

where Q is *query*$(h^{(j)}, A)$ if $h^{(j)}$ is right-recursive, and *query*$(h^{(j)}, h^{(j)})$ otherwise. Here, (a more general version of) L appears as the kth subgoal of r_j, and $N = L$. Suppose that the matching tuples used in the body are *query*(H, B), $b_1, \cdots, b_{m-1}$ respectively. Let θ_0 be the most general unifier of *query*(H, B) and *query*$(h^{(j)}, A)$. Let θ_i be θ_{i-1} composed with the most general unifier of $b_i^{(j)}\theta_{i-1}$ and b_i, for $i = 1, \cdots, k-1$, and let $\theta = \theta_{k-1}$. Thus *query*$(L, N) =$ *query*$(b_k^{(j)}, b_k^{(j)})\theta$. *query*$(H, B)$ holds at stage n. Hence, by the induction hypothesis there is a goal G with leftmost atom H, computed answer substitution ϕ, the right ancestor of H is C, and $B = C\phi$. G can be resolved with r_j, using most general unifier ψ to get a goal $G_1 = b_1^{(j)}\theta_0, \cdots, b_m^{(j)}\theta_0, \cdots$, where $\theta_0 = \phi\psi$ is the computed substitution at G_1. By the induction hypothesis, since b_1 holds at stage n, $b_1^{(j)}\theta_0$ has a subrefutation from G' with computed substitution θ_1. Consider G_2, the node with computed substitution θ_1 with leftmost atom $b_2^{(j)}\theta_1$, at which the previous subrefutation was generated. Again, by the induction hypothesis, there is a subrefutation for $b_2^{(j)}\theta_1$ with computed substitution θ_2. We proceed in this way until we reach a goal node G_k. The computed substitution at G_k is $\theta_{k-1} = \theta$, and the leftmost atom in G_k is $b_k^{(j)}\theta = L$. The right ancestor of L is L itself.

We now consider correspondence (b). Suppose N is inferred at stage $n+1$.

If the predicate of N is not right recursive, then N must have been inferred due to the firing of a rewritten rule

$$h^{(j)} \leftarrow query(h^{(j)}, h^{(j)}), b_1^{(j)}, \cdots, b_m^{(j)}$$

where r_j is a variant of an original rule given by

$$h^{(j)} \leftarrow b_1^{(j)}, \cdots, b_m^{(j)}.$$

Let θ be the substitution generated by the rule firing (as in the proof of correspondence (a) above), so that $h^{(j)}\theta = N$. Each of the conjuncts used in the body must have an appropriate instance that holds at stage n. Using an argument similar to that used for correspondence (a) above, we can find a goal node G_{m+1} such that the following conditions hold: The computed substitution at G_{m+1} is θ, and the leftmost atom in G_{m+1} is not from r_j. The goal node G corresponding to the *query* tuple used in firing the rewritten rule above has leftmost atom L, say, that is its own right ancestor in G. G_{m+1} is a descendent of G in which all atoms "deriving from" L have been resolved out. Hence there is a subrefutation for L with computed answer substitution θ. Finally, by tracing the individual substitutions generated as in correspondence (b) above, we can shown that $N = L\theta$.

If the predicate of N is right recursive, then N must have been inferred due to the firing of a rewritten rule

$$A \leftarrow query(h^{(j)}, A), b_1^{(j)}, \cdots, b_m^{(j)}$$

where r_j is a variant of an original rule given by

$$h^{(j)} \leftarrow b_1^{(j)}, \cdots, b_m^{(j)}.$$

Let θ be the substitution generated by the rule firing (as in the proof of correspondence (a) above), so that $A\theta = N$. Each of the conjuncts used in the body must have an appropriate instance that holds at stage n. Using an argument similar to that used for correspondence (a) above, we can find a goal node G_{m+1} such that the following conditions hold: The computed substitution at G_{m+1} is θ, and the leftmost atom in G_{m+1} is not from r_j. The goal node G corresponding to the *query* tuple used in firing the rewritten rule above has leftmost atom L, say, with right ancestor C, appearing leftmost in G_C. The right ancestor of C is G_C is C itself. G_{m+1} is a (closest) descendent of G_C in which all rule-descendents of C in G have been resolved out. Hence there is a subrefutation for C with computed answer substitution θ. Finally, by tracing the individual substitutions generated as in correspondence (a) above, we can shown that $N = C\theta$. ▮

Corollary 6.4.2: (Correctness) Let P be a Horn program with right-recursive predicates, and let P' be MTRR(P) for the atomic query Q. Then the bottom-up evaluation of P' correctly answers the query Q with respect to the least Herbrand model of P.
Proof: By Theorem 6.4.1, the soundness and completeness of SLD-resolution [7, 28, 45, 60] with respect to the least Herbrand model, and since MTRR(P) and MTRR$'$(P) are equivalent for all EDB and IDB predicates from P. ▮

Note that our correctness result doesn't state that the bottom-up evaluation of MTRR(P) terminates. In general, with recursively applied function symbols, termination is not guaranteed in either SLD-resolution or in the corresponding bottom-up evaluation. However, every correct answer to the query will eventually be generated by the evaluation of MTRR(P). In the case of function-free programs, the evaluation of MTRR(P) is guaranteed to terminate.

6.5 Nonrepetition

When rightmost subgoals of right recursive predicates satisfy the condition of right-linearity, our transformation is essentially the same as one developed by Sagiv [101] for this special case. The idea of "carrying around" a pointer from a subquery to the "top query" was introduced in a simpler form in [49] and (independently) in [73] for right-linear programs with multiple bindings.

While our transformation may sometimes do worse than magic templates, we can show that it does no worse for the class of nonrepeating programs.

Definition 6.5.1: (Nonrepetition) Let P be a program, Q a query, and T the SLD-tree for Q with respect to P. We say P is *nonrepeating* with respect to Q if for every pair of leftmost IDB atoms a and a' in nodes N and N' of T, the following condition holds: If θ and θ' are the computed answer substitutions at N and N' respectively, and h and h' are their respective right ancestors, and if a is an instance of a', then the pair $(a, h\theta)$ is an instance of $(a', h'\theta')$. □

The program of Example 6.2.1 is nonrepeating for the query ?-$p(1, X)$, as are its modified versions in Example 6.2.2 parts (b) and (c). Note that the SLD-tree for Q with respect to P need not be finite in order for P to be nonrepeating with respect to Q.

At the end of this section we shall show that the modified magic templates method does no worse (and often much better) than ordinary magic templates for nonrepeating programs.

So far, the examples where savings over magic sets are achieved have been right linear programs, so that right-linear optimization [76, 77, 113] could have been applied.

Definition 6.5.2: (Right Linearity) Let P be a program in which there is a single IDB predicate p, which is labelled as right-recursive. Let α be an adornment for p. Then P is *right-linear* with respect to α if

1. All rules have at most one occurrence of p in the body.

2. If the head adornment is α, and the recursive rules have their subgoals ordered so that the p subgoal is rightmost, then the resulting adornment for the p subgoal is α.

3. For each recursive rule, each argument of p that is free according to α has the same variable in both the head and the recursive subgoal, and each of these variables appears only in these two positions. □

In fact, it is noted in [113] that Prolog effectively applies tail-recursion optimization for right-linear programs. We shall demonstrate below some examples of nonrepeating programs that are not right linear, for which tail-recursion elimination produces a significant speedup.

Example 6.5.1: Consider the program

$$\begin{array}{l} p(X, Z_1, Z_2) \leftarrow e(X, Y), p(Y, Z_2, Z_1) \\ p(n, Z_1, Z_2) \leftarrow t(Z_1), q(Z_2) \\ q(0) \end{array}$$

This program is not right-linear, due to the fact that the positions of Z_1 and Z_2 are switched in the recursive subgoal of the first rule. It is nonrepeating (for an arbitrary e relation) with respect to a query on p with bound first argument and second and third arguments free. □

Example 6.5.2: The program

$$\begin{array}{l} p(X, t(Y), Z) \leftarrow e(X, W), p(W, Y, s(Z)) \\ p(X, Y, Z) \leftarrow f(X, Y, Z) \end{array}$$

is nonrepeating (for arbitrary e and f relations) with respect to a query on p with free third argument and first and second arguments bound. This program is not right-linear due to the function symbol around Z in the body of the first rule. □

Example 6.5.3: Let P be

$$\begin{aligned}
&leaf(t(T_1, T_2), s(L)) \leftarrow leaf(T_1, L)\\
&leaf(t(T_1, T_2), s(L)) \leftarrow leaf(T_2, L)\\
&leaf(L, s(L)) \leftarrow atomic(L)
\end{aligned}$$

Given a binary tree T_0, $leaf(T_0, s^i(L))$ holds for each leaf L of T_0 having depth i.[5] Consider the performance of magic templates with right-recursion for the query ?-$leaf(t_0, L)$, where t_0 is a complete tree of depth d having $n = 2^d - 1$ nodes and $l = 2^{d-1}$ leaves. There will be $O(n)$ *query* tuples. While the size of the tuples themselves add up to give an extra factor of d to the total size, clever management of the tree structure using pointers can reduce this factor to a constant.[6] There will be l *leaf* answer tuples. Thus, semi-naive evaluation will take $O(n)$ time and space.

By comparison, magic sets will generate *leaf* tuples at every level of the tree, generating a total of ld *leaf* tuples. Thus magic sets will have time and space complexity $\Omega(n \log n)$. □

Example 6.5.4: Let the relation *subtree* encode a binary tree,[7] so that $subtree(t_0, t_1, t_2)$ holds when t_1 is the left subtree of t_0 and t_2 is the right subtree of t_0. (If one subtree is missing, we substitute the constant symbol *null*.) We say that a subtree rooted at t is *balanced* if t has no descendents with exactly one child. Then the program

$$\begin{aligned}
&p(R, X) \leftarrow subtree(R, R_1, R_2), p(R_1, Z), p(R_2, X)\\
&p(R, X) \leftarrow atomic(R), values(R, X)
\end{aligned}$$

is nonrepeating, assuming that *atomic* holds for all leaves with respect to *subtree*, and not for *null*. $p(r, X)$ succeeds if the subtree rooted at r is balanced, and has corresponding rightmost *values* X. □

The crucial feature of all the examples above for which top-down does better, is that the rightmost subgoal shares at least one unbound variable with the head. We can effectively say, "All solutions for the free variables in the rightmost subgoal are solutions for the head," *without explicitly representing the solution of the rightmost subgoal.*

Theorem 6.5.1: (Efficiency) Let P be a program, and let Q be a query on a predicate from P such that P is nonrepeating with respect to Q. Then evaluating the program $P_R = \mathrm{MTRR}(P)$, resulting from the magic templates rewriting with right-recursion for P and Q, performs no more inferences than the program P_T generated by magic templates for P and Q.
Proof: By Theorem 6.4.1, $query(L, N)$ holds when there is a goal G with leftmost atom L, right ancestor M and $N = M\theta$, where θ is the computed substitution at G. The *magic* tuples generated by P_T are either magic tuples generated by P_R, or correspond to first arguments of *query* tuples generated by P_R. The property of nonrepetition guarantees that the extra argument A in both the *query* and *sup* tuples does not affect duplicate elimination in P_R compared with P_T, and hence the number of tuples generated for these predicates. Finally, every time a rule from P_R of the form

$$A \leftarrow sup_{j.k}(\vec{Z}, A), q(\vec{Y})$$

[5]We assume all leaves are distinct, so that the structure is indeed a tree and not a directed acyclic graph.

[6]This same observation about pointers could also be used in the regular magic-templates construction, and so it does not affect the comparison made here.

[7]Again, we assume no common subtrees.

fires, then a corresponding rule of the form

$$p(\vec{X}) \leftarrow sup_{j.k}(\vec{Z}), q(\vec{Y})$$

from P_T also fires. ∎

The evaluation of P_R may compute significantly fewer tuples, since it does not represent intermediate answers for right-recursive predicates. There is, of course, the overhead of having an extra argument in the *sup* tuples. At worst, this would add a small constant factor to the overall cost.

With the extension proposed above we can do better than "sip-optimal" [88] (See Definition 5.5.1). Modulo the *magic* tuples, magic templates is sip-optimal [88]. Modulo the *magic* and *query* tuples, our extension is at worst sip-optimal for nonrepeating programs, by Theorem 6.5.1. We can do better in some cases, for example the program of Example 6.3.1, because not all intermediate answers need to be represented.

6.6 Weak Right Linearity

We present a syntactically recognizable sufficient condition for a linear program/query pair to be nonrepeating. We call this condition "weak right linearity," since it is satisfied by all right linear programs. The class of weakly right linear programs is more general than the class of right linear programs in three ways:

1. Multiple mutually recursive predicates are permitted.

2. Function symbols are permitted in terms containing free variables.

3. Free variables do not have to be in identical positions in both the head and the right-recursive subgoal.

We adorn instances of variables as either "bound," *b*, or "free," *f*.[8]

Definition 6.6.1: (Weak right linearity) Let P be a range-restricted program, and let $p_1, \ldots, p_n$ be predicates that we label right-recursive. Suppose that at most one such p_i appears in each rule body of P, and that every rule with a right-recursive predicate in the body also has a right-recursive predicate in the head. Let Q be a query on some p_j, with binding pattern α_j. We say P is weakly right linear with respect to Q if, once the rules of P have their subgoals reordered so that the right-recursive predicate (if present) is rightmost,

1. Each p_i has a unique induced binding pattern α_i, and

2. If the rightmost subgoal of a rule is right-recursive then every free variable in the head of that rule appears in the head with no enclosing function symbols, and also appears free (and only free) in the rightmost subgoal, according to its binding pattern. □

[8] We assume that all variables in a bound term are bound, and that a term is free if it contains a free variable.

That right linear programs are weakly right linear is straightforward from the definitions. The programs of Examples 6.5.1 and 6.5.2 are weakly right linear but not right linear. The program of Example 6.5.3 is not weakly right linear due to the presence of the function symbol s in the head of the right-recursive rules. Note that Example 6.5.3 may not be nonrepeating when the initial "tree" contains repeated subtrees, and is thus a directed acyclic graph. Since the nonrepetition of Example 6.5.3 depends upon the form of the query, it will be impossible to recognize that it is nonrepeating using a sufficient syntactic condition. A similar observation holds for Example 6.5.4, where nonrepetition depends upon the values in an EDB relation.

Lemma 6.6.1: Let P be a program, and Q a query. If P is weakly right linear with respect to Q then P is nonrepeating with respect to Q.
Proof: By weak right linearity, all instances of right-recursive predicates will have the same right ancestor, namely the original query, possibly instantiated in different ways. However, since none of the free arguments in the head can have been bound (again, by weak right linearity) the right ancestor will be the uninstantiated query itself, and nonrepetition follows. ∎

Our method yields the same performance as the right-linear transformation for right-linear programs, but is more generally applicable.

6.7 Discussion

We have discussed the question of tail-recursion optimization, motivated by the observation that in some cases SLD-resolution can outperform magic sets by not representing intermediate answers explicitly. We have presented a modification of magic templates that takes advantage of tail-recursion elimination. We have demonstrated the correctness of this rewriting, and have shown that this method does at least as well (and often better) than magic templates for the class of nonrepeating programs. We have provided a syntactic sufficient condition for nonrepetition, namely weak right-linearity. The class of weakly-right-linear programs properly includes the class of right-linear programs.

As noted by Sagiv [101], the class of left-linear rules [77] can be generalized in a symmetric way. Left-linear optimization and mixed-linear optimization [77] can then be suitably generalized for programs with rules that commute and are either weakly right-linear or "weakly left-linear."

Chapter 7

Modular Acyclicity

7.1 Introduction

We now consider the issue of acyclicity, i.e., the absence of both positive and negative loops in logic programs or deductive databases. In the absence of loops, some desirable semantic properties hold, and certain program optimizations become possible.

We generalize the class of "acyclic" programs [5, 20] to what we call "modularly acyclic" programs. Acyclic programs P have the following desirable properties:

- They are locally stratified, and hence have a unique perfect model M_P.
- M_P is the unique Herbrand model of P's completion.
- Membership in M_P is decidable.
- For bounded goals, SLS-resolution and SLDNF-resolution coincide. For bounded goals that do not flounder, SLDNF resolution is sound and complete with respect to M_P, and always terminates.

We show that modularly acyclic programs satisfy analogous properties, while being significantly more expressive than acyclic programs. For modularly acyclic Datalog programs, semi-naive bottom-up evaluation will generate a tuple only finitely many times if we don't do duplicate elimination. Since duplicate elimination is therefore not necessary for termination, we can choose to discard or defer duplicate elimination in order to speed up query evaluation. As remarked in [64] the cost of duplicate detection can be a large proportion of the total cost of query evaluation. While discarding duplicate elimination may actually slow down query evaluation in some cases, one can use an adaptive duplicate elimination strategy that eliminates duplicates only "when necessary."

While we show that both modular stratifiability and modular acyclicity are undecidable properties of programs, there are situations where modular acyclicity can be demonstrated quite easily using the techniques of monotonicity constraints [15].

7.2 Modular Acyclicity

We shall generalize the class of acyclic programs to the class of modularly acyclic programs, in the same way that local stratification is generalized to modular stratification in Section 5.2.

Definition 7.2.1: A *level mapping* is a mapping from ground atoms to the natural numbers $\mathbf{N}$. We write $\|a\|$ for the level of the ground atom a, and extend the level mapping to ground literals according to $\|\neg a\| = \|a\|$. $\Box$

Note that in this chapter, a level mapping is to the natural numbers, and not to countable ordinals.

The following definition of acyclic programs is from [5]. In [20] they are termed ω-locally hierarchical programs.

Definition 7.2.2: A program P is *acyclic* if there is some level mapping such that for every instance of each rule in P, the level of the atom in the head is strictly greater than the level of each literal in the body. $\Box$

Recall that a *component* F of a program P is a maximal subset of the rules of P such that all predicates appearing in the head of rules in F are mutually recursive, possibly through negation. We write $F_1 \sqsubset F_2$ if there is a predicate in the body of some rule in F_2 that is the head of a rule in F_1. Define $\prec$ to be the transitive closure of $\sqsubset$, as in Section 5.2. We use the concept of the reduction of a component (Definition 5.2.3).

Definition 7.2.3: A program P is *modularly acyclic* if for every component F of P,

- There is a total well founded model M for the union of all components $F' \prec F$, and
- The reduction of F modulo M is acyclic. $\Box$

Since every acyclic program is locally stratified, it follows that every modularly acyclic program is modularly stratified.

Definition 7.2.4: An atom $p(\vec{X})$ is *bounded* with respect to a level mapping if the set of levels of all instances of $p(\vec{X})$ is finite. $\Box$

7.2.1 Semantic Properties

There has been much recent work on the semantics of logic programs with negation. The class of modularly acyclic programs has the remarkable property that all the major proposals agree on their semantics. The completion-based approaches, in general, differ from the other approaches for programs that have positive loops. For example, the program

$$p \leftarrow p$$

does not entail $\neg p$ according to the program completion. However such positive loops are outlawed by the condition of acyclicity, so that this semantic difference will not arise in our context.

Another interesting consequence of a program's acyclicity is that SLDNF-resolution terminates for bounded goals, since all SLD-trees are finite. We shall extend these results (together with some other results) to the class of modularly acyclic programs.

The following results are summarized in [5].

Theorem 7.2.1: Let P be an acyclic program. Then

1. T_P has a unique fixpoint, M_P.
2. M_P is the perfect model of P.
3. M_P is the unique Herbrand model of $comp(P)$.
4. M_P is the unique fixpoint of $\mathbf{N}_P$.
5. For all ground atoms A that do not flounder, $A \in M_P$ iff there exists an SLDNF-refutation for the query ?-A with respect to P.
6. For all ground atoms A that do not flounder, $A \in M_P$ iff there exists an SLS-refutation for the query ?-A with respect to P.
7. Membership in M_P is decidable. ▮

Here, T_P is the two-valued immediate consequence operator defined in [60]. $\mathbf{N}_P$ is the three-valued immediate consequence operator of Fitting [34] (See Definition 3.4.5).

Our main semantic results are summarized in the following theorem.

Theorem 7.2.2: Let P be a modularly acyclic program. Then

1. T_P has a unique fixpoint, M_P.
2. M_P is the well-founded model of P, which is two-valued.
3. M_P is the unique Herbrand model of $comp(P)$.
4. M_P is the unique fixpoint of $\mathbf{N}_P$.
5. For all ground atoms A that do not flounder, $A \in M_P$ iff there exists an SLDNF-refutation for the query ?-A with respect to P.
6. For all ground atoms A that do not flounder, $A \in M_P$ iff there exists an SLS-refutation for the query ?-A with respect to P.
7. Membership in M_P is decidable.

Proof: (Sketch) The proofs are by induction on the dependency relation between program components using Theorem 7.2.1 at each step. Part 2 follows because modularly acyclic programs are modularly stratified. The proof of parts 5 and 6 assume that the literals in the body of every rule are ordered with subgoals from the same component as the head predicate appearing to the right of the other subgoals. ▮

The class of acyclic programs is relatively limited. For example, the transitive closure program

$$
\begin{array}{l}
tc(X,Y) \leftarrow e(X,Y) \\
tc(X,Z) \leftarrow e(X,Y), tc(Y,Z) \\
e(a,b) \\
\vdots
\end{array}
$$

is not an acyclic program, even if there are no cycles in either tc or e. However, if e is a finite acyclic relation, then the above program is modularly acyclic. The reason that the program above is not considered acyclic according to the definitions of [5] is that the instantiation

$$tc(a,a) \leftarrow e(a,a), tc(a,a)$$

of the second rule precludes any level mapping, since $tc(a,a)$ would have to be of lower level than itself. In the case where e is known to be acyclic, considering the instantiation above is unnecessary. The condition of modular acyclicity considers only those rule instantiations whose subgoals from lower components are known to be true, and thus admits a significantly larger number of programs.

7.2.2 Duplicate Elimination

Bottom-up query evaluation methods typically involve rewriting the program so that semi-naive bottom-up evaluation of the rewritten rules (with duplicate elimination) is more efficient than evaluating the original rules directly. The biggest risk of omitting duplicate elimination in bottom-up methods is nontermination. For example, the program

$$\begin{array}{l} p(X) \leftarrow e(X,Y), p(Y) \\ p(a) \\ e(a,a) \end{array}$$

would continue generating the tuple $p(a)$ indefinitely. However, for modularly acyclic programs tuples can only be generated a finite number of times, and so for Datalog programs bottom-up evaluation would still terminate. Modular acyclicity is a sufficient condition for the "finite forest" property [64]. As observed in [64] the cost of duplicate detection and elimination can be a large proportion of the total cost of query evaluation.

Magic-sets is a widely employed technique for rewriting programs for efficient bottom-up evaluation [11, 13, 88]. Rewriting a modularly acyclic program using magic set techniques does not necessarily yield a modularly acyclic program. Hence one may ask whether tuples are generated only finitely often when evaluating the magic-rewritten program. Mumick et al. [74] have considered using duplicates as semantically meaningful entities, where the number of copies of a tuple corresponds to the number of derivations of that tuple. While their motivation is different from ours, they show that (in the case of range restricted programs) the multiplicity of tuples computed by a version of semi-naive evaluation corresponds exactly to the number of top-down "proof trees" for that tuple *if duplicate elimination is applied to the magic predicates only.* Thus, for modularly acyclic programs we can evaluate the magic rewritten program and terminate by eliminating duplicates on the magic predicates only.

Since the multiplicity of any tuple is finite for a modularly acyclic program, we can formulate an adaptive duplicate elimination strategy: Eliminate duplicates only "when necessary." There are several possible techniques that could be employed. One strategy would be to eliminate duplicates only when a relation becomes unbearably large. Another would be to eliminate duplicates when a random sampling from the relation indicates a relatively high duplicate frequency. Alternatively, we could eliminate duplicates "at join time," when a pass is made through the relation anyway, rather than after each insertion.

7.2.3 Decidability

In the case where function symbols are allowed, testing for local stratification or acyclicity is undecidable [26]. In the function-free case, however, it is decidable whether a given program is locally stratified or acyclic.

One may ask, in a database context, whether a property holds of a program *for all possible relations assignable to EDB predicates.* For example, two programs may be said to be equivalent if they compute the same relation as one another no matter what tuples are in the EDB relations.

We shall ask similar questions of modular stratification and modular acyclicity, i.e., whether a given set of rules is modularly acyclic or modularly stratified for all EDB relations satisfying certain constraints. We show that this problem is undecidable even for function-free programs.

In our context we will not necessarily want to consider all possible EDB relations. For example, if *part*(X, Y) represents a relation describing the sub-parts of a part, we are only really interested in those relations for *part* that are acyclic. If *part* is known to be acyclic then we may be able to prove modular acyclicity of the whole program.

Definition 7.2.5: (Acyclicity Constraint) Let e be a relation of arity at least two, and let a_1 and a_2 be distinct argument positions of e. We say e is *acyclic* if the projection of e onto a_1 and a_2, considered as a set of directed edges between argument values, is an acyclic graph. An *acyclicity constraint* is a requirement that a given relation be acyclic for a given pair of argument positions. □

Thus we shall pose the statement of the test as follows: "Is P modularly acyclic (modularly stratified) for all EDB's satisfying a given set of acyclicity constraints on the EDB predicates." The main results of this section are that modular stratification and modular acyclicity are both undecidable properties of programs. To establish these results we need some preliminary results The following lemma makes use of a construction of Shmueli that is reviewed in Section 7.3.

Lemma 7.2.3: It is recursively unsolvable to determine, for an arbitrary Datalog program whether a given relation is transitively closed for all possible assignments of relations to EDB predicates.

Proof: We first show that testing whether a context free language L (not including the empty string[1]) has the property that $LL \subseteq L$ is undecidable. Let L be the context-free language generating the invalid computations of a Turing machine M (minus the empty string) constructed using standard techniques as in [46]. Then $LL \subseteq L$ if and only if $L(M) = \emptyset$, since if there were something in $L(M)$ then the corresponding valid computation could be split into two nonempty invalid computations, implying $LL \not\subseteq L$. Since detemining for a given Turing machine M whether $L(M) = \emptyset$ is undecidable, it follows that determining whether $LL \subseteq L$ is undecidable for context-free languages L (not containing the empty string).

Let $G = (N, T, P, S)$ be a context-free grammar with nonterminals N, terminals T, productions P and starting nonterminal S. Let G' be the context free grammar

$$(N \cup \{R\}, T, P \cup \{R \rightarrow SS, R \rightarrow S\}, R)$$

where R does not appear in N or T. Clearly $L(G)L(G) \subseteq L(G)$ if and only if $L(G') \subseteq L(G)$.

[1]We make the technical restriction that our context-free languages do not contain the empty string for compatibility with Shmueli's construction that is used later.

Consider $Q(G)$ and $Q(G')$ as in Appendix 7.3, and let ?-$g(I,J)$ and ?-$g'(I,J)$ be their respective queries. ($Q(G)$ and $Q(G')$ are both Datalog programs.) By Lemma 7.3.1 it follows that $L(G') \subseteq L(G)$ if and only if g' is transitively closed for all EDB's. Since the problem of determining whether $L(G)L(G) \subseteq L(G)$ is undecidable, it follows that the problem of determining whether a predicate is transitively closed for all values of EDB predicates is also undecidable. ▮

Theorem 7.2.4: It is recursively unsolvable to determine, for an arbitrary function-free program whether a program is modularly stratified for all EDB's.
Proof: Consider a program P containing the rule

$$p(X) \leftarrow t(X,Y), t(Y,Z), \neg t(X,Z), \neg p(X)$$

together with some Datalog rules defining the IDB predicate t, but not mentioning p. Then P is modularly stratified if and only if t is transitively closed. Hence determining whether a program is modularly stratified for all values of EDB predicates is undecidable, by Lemma 7.2.3. ▮

Theorem 7.2.5: It is recursively unsolvable to determine, for an arbitrary program and an arbitrary acyclicity constraint whether a program is modularly acyclic for all EDB's satisfying the constraint.
Proof: The proof is similar to that of Theorem 7.2.4, except some additional details are needed to make the result of the construction modularly acyclic. Every context-free language not containing the empty string can be generated by a context-free grammar in Greibach normal form [41]. If G is a context-free grammar in Greibach normal form, then $Q(G)$ has the property that every rule (apart from the one with head matching the query) has a subgoal of the form $l(I_0, A_j, I_1)$.

We modify the construction of $Q(G)$ as follows to get a new program $Q'(G)$:

- Include the two rules
$$tc(I_0, I_1) \leftarrow l(I_0, A, I_1)$$
$$tc(I_0, I_2) \leftarrow l(I_0, A, I_1), tc(I_1, I_2)$$
where tc occurs nowhere else in $Q(G)$, so that tc is the transitive closure of the projection of l on its first and third arguments.

- Whenever a subgoal of the form $r(I_{k-1}, I_k, A_1, \cdots, A_m)$ (corresponding to a nonterminal symbol on the right hand side of a production in G) appears in the body of a rule, append the additional subgoal $tc(I_{k-1}, I_k)$.

Note that the answers to the queries on $Q(G)$ and $Q'(G)$ are the same, since it can easily be shown by induction on the bottom-up evaluation of the program of $Q(G)$ that for all true atoms of the form $r(I_{k-1}, I_k, A_1, \cdots, A_m)$, $tc(I_{k-1}, I_k)$ must hold.

We place an acyclicity constraint on l that it must be acyclic in its first and third arguments. Let $<_l$ be a partial order on the third and first arguments of l, so that if $l(a,b,c)$ holds then $c <_l a$. By the construction of the rules for tc, it follows that tuples of the form $tc(a,b)$ satisfy $b <_l a$. With this constraint, we can be sure that $Q'(G)$ is modularly acyclic, since once the rules have been reduced modulo the well-founded model for the l and tc rules, the first argument of every predicate in the body is smaller than the first argument in the head of every subgoal according to $<_l$.

Let P be a modularly acyclic Datalog program not mentioning the predicate p. Then P together with the rule

$$p(X) \leftarrow t(X,Y), t(Y,Z), \neg t(X,Z), \neg p(X)$$

is modularly acyclic if and only if t is transitively closed in P. It is undecidable whether t is transitively closed for all EDB's satisfying the acyclicity constraint on l, by an argument similar to that of Lemma 7.2.3, but using Lemma 7.3.2 rather than Lemma 7.3.1.

It then follows that determining whether a program is modularly acyclic is undecidable, since we can choose P to be $Q'(G)$, which is modularly acyclic, rather than $Q(G)$. ∎

The proof of Theorem 7.2.4 makes no use of acyclicity constraints, and so modular stratification is certainly undecidable when acyclicicty constraints can be specified. The proof of Theorem 7.2.5 does use one acyclicity constraint, and undecidability therefore follows for multiple acyclicity constraints.

Modular stratification and modular acyclicity for all EDB's is undecidable even in the function-free case. However, sufficient conditions for modular acyclicity and modular stratification can be established using monotonicity constraints [15]. A monotonicity constraint is similar to an acyclicity constraint, except that a partial order on domain values is specified in advance. More than one partial order may be necessary, and inferences about each partial order must be made independently, since they may not be mutually compatible.

For example, suppose we are given the rules

$$\begin{aligned} p(X,Y) &\leftarrow e(X,X'), f(Y,Y'), \neg t(X,Y), p(X',Y') \\ p(X,Y) &\leftarrow f(Y,Y'), p(X,Y'). \end{aligned}$$

Suppose we know that the tuples $f(Y,Y')$ satisfy $Y' < Y$, where $<$ is some partial order on the domain of the database. We could ignore the negative subgoal, and establish that cycles do not occur in these rules since the second argument of p is decreasing with respect to $<$ at each step.

7.3 Shmueli's Construction

In this section we review the construction of Datalog programs from context-free grammars from [109].

Let $G = (N, T, P, S)$ be a context free grammar, where N is a finite set of nonterminals, T is a finite set of terminals, P is a finite set of productions and S is the start symbol. $N \cap T = \emptyset$ and $S \in N$. Let $T = \{a_1, \cdots, a_n\}$. We shall restrict attention to context-free grammars that contain no empty productions, and such that S does not appear on the right hand side of any production. Any context-free grammar (except those producing the empty string) can be transformed into this form.

Define $Q(G)$ to be a query-program pair as follows:

The query is ?-$g(I,J)$. One rule in $Q(G)$ is

$$g(I,J) \leftarrow h(A_1, \cdots, A_n), s(I, J, A_1, \cdots, A_n).$$

The remaining rules are constructed in one-to-one correspondence with the productions of G. Let

$$V \rightarrow b_1 \cdots b_m$$

be a production in G, where $V \in N$, and $b_j \in (N \cup T)$ for $j = 1, \ldots, m$. The corresponding rule is

$$v(I_0, I_m, A_1, \cdots, A_n), \leftarrow C_1, \cdots, C_m$$

where for $k = 1, \ldots, m$, C_k is as follows: If b_k is a nonterminal, say W, then C_k is $w(I_{k-1}, I_k, A_1, \cdots, A_n)$. Otherwise, b_k is a terminal symbol, say a_q, and C_k is $l(I_{k-1}, A_q, I_k)$.

Example 7.3.1: Let $G = (N, T, P, S)$, with $T = \{a_1, a_2, a_3\}$, $N = \{Q, R, S\}$, and $P = \{S \rightarrow Q, Q \rightarrow a_1 R a_2, R \rightarrow a_1 a_1 Q, R \rightarrow a_2 a_3\}$. $Q(G)$ is then

$$\begin{array}{l}
? - g(I, J) \\
g(I, J) \leftarrow h(A_1, A_2, A_3), s(I, J, A_1, A_2, A_3) \\
s(I_0, I_1, A_1, A_2, A_3) \leftarrow q(I_0, I_1, A_1, A_2, A_3) \\
q(I_0, I_3, A_1, A_2, A_3) \leftarrow l(I_0, A_1, I_1), r(I_1, I_2, A_1, A_2, A_3), l(I_2, A_2, I_3) \\
r(I_0, I_3, A_1, A_2, A_3) \leftarrow l(I_0, A_1, I_1), l(I_1, A_1, I_2), q(I_2, I_3, A_1, A_2, A_3) \\
r(I_0, I_2, A_1, A_2, A_3) \leftarrow l(I_0, A_2, I_1), l(I_1, A_3, I_2)
\end{array}$$

□

Let Q_1 and Q_2 be query-program pairs. We shall write $Q_1 \subseteq Q_2$ if for every assignment of values to the EDB relations in Q_1 and Q_2, the answer to the query of Q_1 is contained in the answer to the query of Q_2.

Lemma 7.3.1: Let G_1 and G_2 be context-free grammars not producing the empty string. Then $Q(G_1) \subseteq Q(G_2)$ if and only if $L(G_1) \subseteq L(G_2)$.
Proof: See [109]. ▮

In Section 7.2.3 we need a slight variant of this lemma when l is constrained to be acyclic. Let Q_1 and Q_2 be query-program pairs. Let us write $Q_1 \subseteq_C Q_2$ to mean that for every assignment of values satisfying the acyclicity constraint(s) C to the EDB relations in Q_1 and Q_2, the answer to the query of Q_1 is contained in the answer to the query of Q_2.

Lemma 7.3.2: Let G_1 and G_2 be context-free grammars not producing the empty string. Let C be the acyclicity constraint specifying that the EDB relation l is acyclic in its first and third arguments. Then $Q(G_1) \subseteq_C Q(G_2)$ if and only if $L(G_1) \subseteq L(G_2)$.
Proof: The proof in [109] goes through even with the restriction that l is acyclic. ▮

Chapter 8

Semantics for Second Order Programs

A number of "higher order" logics for programming languages and database systems have been recently proposed. One of these proposals is HiLog [25]. HiLog provides a declarative framework for incorporating second order features into logic programs. HiLog has also been chosen by the NAIL! group at Stanford University as the basis for its deductive database system. A distinguishing feature of HiLog is that it has a "first order semantics" even though its syntax is second order. The crucial feature that reduces the semantics to first order is that nested relations are modeled by having relation *names* as arguments rather than *the relations themselves* as arguments.

HiLog allows arbitrary terms to be relation names. Hence, in the formal development, there is no distinction between function, constant and predicate symbols. In rules, variables may appear not only in argument positions, but also in predicate names.

In [25] the semantics of HiLog is defined, and it is shown that unification is decidable and that resolution is both sound and complete for HiLog. When HiLog literals are allowed in rules, we call the resulting programs "HiLog programs." One way to understand the semantics of HiLog programs is via a "universal relation model." Imagine a "universal relation" u_i of each arity i, and transform each n-ary HiLog atom to an $(n+1)$-ary atom involving the corresponding universal predicate. For example, the HiLog atom $p(X,a)(Z)$ would be rewritten as

$$u_2(p(X,a),Z).$$

The semantics of a HiLog program may then be interpreted as the semantics of its "universal version."

The question addressed in this paper is the semantics of HiLog programs when negation is allowed in the rule bodies. For normal programs there has been much recent work on semantics in the presence of negation in the rule bodies. Stratification is a syntactic restriction on programs that prevents a predicate from recursing negatively through itself [6, 24, 116]. A stratified program has a well defined semantics given by the Herbrand model constructed by taking least fixpoints at successively higher "levels."

However, there are nonstratified programs for which a clear semantics can be discerned. This observation led to the development of the well-founded semantics [118] and the stable-model semantics [37] which provide semantics for programs that are not necessarily stratified. (See Chapter 3.)

In this chapter we generalize the well-founded semantics and stable model semantics to HiLog programs that can have negation in the rule bodies. In order for the HiLog semantics to reduce to the semantics for normal programs when the HiLog program happens to be normal, we observe that domain independence is a critical condition, and show that the semantics do in fact reduce in this way for range restricted programs.

We discuss the relationship between domain independence and what we term "preservation under extensions." Informally, a semantics for a program P is preserved under extensions if adding some extra ground rules to P that mention no symbols originally in P does not change the semantics of the predicates originally in P. For normal programs, domain independence and preservation under extensions coincide for both the well-founded and stable model semantics. However, for HiLog programs, preservation under extensions is a strictly stronger condition. We argue that preservation under extensions is a desirable property for HiLog programs. We generalize the concept of range restrictedness to HiLog programs, and show that range-restrictedness is a sufficient condition for preservation under extensions with respect to the well-founded semantics. A slightly stronger condition, which we call "strong range-restrictedness" is a sufficient condition for preservation under extensions with respect to the stable model semantics.

We then discuss under what conditions can we be sure that the well-founded semantics (and hence the stable model semantics) described above are two-valued, i.e., make nothing undefined. Stratification is such a condition, although not the most general. In any case, with variables in predicate names, defining levels on predicates can be problematic.

In Chapter 5 we described the class of "modularly stratified" (normal) programs. Modular stratification ensures a two-valued semantics while allowing a broader class of programs than just the stratified ones. We generalize modular stratification to HiLog programs. (This generalization cannot make use of the universal relation model described above, since the universal relation transformation does not preserve the strongly connected components of the program.) We show that modularly stratified HiLog programs have a two-valued well-founded semantics.

We show how to generalize the techniques of Section 5.4.4 to obtain a magic sets method for modularly stratified HiLog programs. This rewriting method allows the efficient evaluation of queries over a large class of HiLog programs.

HiLog has applications in logic programming where, for example, it provides a declarative way to represent the *call* meta-predicate of Prolog. HiLog has also been chosen by the NAIL! group at Stanford University as the logic on which to base both their declarative and procedural database languages. HiLog has a satisfying (and easily computed) way of handling nested relations and parameterized procedures.

8.1 HiLog

HiLog programs allow predicate names to appear as arguments of other predicates. In fact arbitrary terms may appear as predicate names. Because of this syntax, there is no distinction in HiLog between predicate, function and constant symbols. Hence we simply refer to HiLog "symbols." Correspondingly, there is no distinction between HiLog terms and HiLog atoms; we shall use these interchangeably. In particular, the Herbrand base and the Herbrand universe coincide in the HiLog context.

Definition 8.1.1: The set of *HiLog terms* is defined according to the rules:

- Every symbol is a term, as is every variable.

- For each $n \geq 0$, if $t_1, \cdots, t_n$ and t are terms then so is $t(t_1, \ldots, t_n)$.[1]

If $t(t_1, \ldots, t_n)$ is a term then we say that t is the *name* of the term, and each t_i is an *argument* of the term. A *HiLog literal* is either a HiLog term or a negated HiLog term. A *HiLog rule* is a logical rule of the form

$$A \leftarrow L_1, \ldots, L_n$$

where A is a HiLog term, $n \geq 0$, and each L_i is a HiLog literal. A *HiLog program* is a finite set of HiLog rules. □

In this chapter, where the meaning is clear, we may omit the word "HiLog" from the above definitions. We use the adjective "normal" to distinguish the concepts of normal programs from those of HiLog programs.

Example 8.1.1: [25] The following HiLog program implements the Lisp-style *maplist* operation.

$$\begin{array}{l} \mathit{maplist}(F)([],[]) \\ \mathit{maplist}(F)([X|R],[Y|Z]) \leftarrow F(X,Y), \mathit{maplist}(F)(R,Z) \end{array}$$

One calls the predicate $\mathit{maplist}(f)$ (for some ground term f), with lists as arguments. $\mathit{maplist}(f)$ succeeds, possibly binding variables in its list arguments, when f succeeds pairwise on the list elements. With normal logic programs one would have to write a separate *maplist* routine for each possible f, rather than one generic routine. □

The semantics of HiLog programs (without negation in the body) is first order. To understand the semantics of HiLog one can imagine transforming a HiLog program into an normal program having a generic predicate name u_i of each arity i. For example, the program of Example 8.1.1 would become

$$\begin{array}{l} u_3(\mathit{maplist}(F),[],[]) \\ u_3(\mathit{maplist}(F),[X|R],[Y|Z]) \leftarrow u_3(F,X,Y), u_3(\mathit{maplist}(F),R,Z) \end{array}$$

This program is Horn, and when interpreted with respect to the HiLog Herbrand universe, it determines a least model that is taken to be the semantics of the program.[2] We refer to this transformation into generic predicates as the "universal relation model."

The Herbrand universe of a program depends only upon the symbols appearing in the program and not on their arities. Also there is no distinction between predicate, function and constant symbols. We say that the Herbrand universe is *generated* by the symbols appearing therein. From those symbols, all possible terms of all arities can be constructed. The HiLog Herbrand universe will always be a countably infinite set.

Since we shall be dealing with a 3-valued logic, we shall define an *Herbrand interpretation* to be a consistent set of ground (HiLog) literals. If neither an atom nor its negation is

[1] In [25] the restriction $n \geq 1$ is made, excluding 0-ary predicates (but without losing any expressive power). In this paper, we distinguish between variables appearing in predicate names and those appearing in arguments, and so we shall allow 0-ary predicates. We may need to disambiguate terms like $p(3)$, an instance of the unary predicate p, from $p(3)$, the 0-ary predicate name. When this is necessary, we shall write the 0-ary atom with name $p(3)$ as $p(3)()$.

[2] Strictly speaking, one needs universal function symbols too. Our presentation of the universal relation model is for motivational purposes only, and does not affect any of our results.

in the interpretation I, then that atom is *undefined* with respect to I. Since we shall be dealing exclusively with Herbrand interpretations throughout this paper, we shall omit the term "Herbrand" for readability; we shall also omit the mapping from constants, function symbols and predicates to domain elements, functions and relations, respectively, since all symbols are interpreted freely. A *total* interpretation is an interpretation in which nothing is undefined.

Definition 8.1.2: An Herbrand interpretation I is a quadruple $< P_I, F_I, C_I, S_I >$, where P_I is a set of predicate names each with an associated set of arities, F_I is a set of function names each with an associated set of arities, C_I is a nonempty set of constant symbols, and S_I is a consistent set of ground literals that can be formed using predicates, functions and constants from P_I, F_I and C_I respectively.[3] We may write "$p \in I$" as shorthand for "$p \in S_I$". $\Box$

Normal interpretations have separate sets of predicate function and constant names, each of which is a single symbol, and with predicate and function symbols having a single arity. HiLog interpretations I have both P_I and F_I equal to the set of all variable-free HiLog terms, each HiLog term having arity set equal to $\{0, 1, 2, \ldots\}$. Also, for HiLog interpretations C_I is just the set of (non-variable) symbols in the program.

Definition 8.1.3: Let I and I' be interpretations. We say I is *upward compatible* with I' if $P_I \subseteq P_{I'}$, $F_I \subseteq F_{I'}$, $C_I \subseteq C_{I'}$, and the arities of each p in P_I and f in F_I are also arities of p in $P_{I'}$ and f in $F_{I'}$ respectively. $\Box$

A normal interpretation of a normal program will be upward compatible with every HiLog interpretation for the same program.

Definition 8.1.4: Let I and I' be interpretations such that I is upward compatible with I'. We say I' *extends* I if the following conditions hold for all atoms p in the language of I' whose name is in P_I.

- $p \in I \Rightarrow p \in I'$
- $\neg p \notin I \wedge (p \text{ is a legal atom in the language of } I) \Rightarrow \neg p \notin I'$.

Informally, I' extends I if everything true in I is also true in I', and everything that is undefined in I is either true or undefined in I'.

We say I' *conservatively extends* I if the following conditions hold for all atoms p in the language of I' whose name is in P_I.

- $p \in I \iff p \in I'$
- $\neg p \notin I \wedge (p \text{ is a legal atom in the language of } I) \iff \neg p \notin I'$.
- $\neg p \in I \Rightarrow \neg p \in I'$.

In other words, I' conservatively extends I if the only extra information in I' about predicates in I is negative, i.e., consists only of negated atoms. It is not important whether I' is true for predicates not constructible from the symbols in I. $\Box$

[3] Of appropriate arities.

8.2 Semantics for HiLog Programs with Negation

8.2.1 Normal Programs

We shall be referring to the well-founded semantics and stable model semantics as described in Chapter 3. Rather than use the original definition of what constitutes a stable model of a program (Definition 3.5.2), we shall use the formulation based on $\mathbf{W}_P$ that is shown to be equivalent in Corollary 3.5.3.

Definition 8.2.1: Let P be a normal program. A *stable model* of P is a two-valued fixpoint of $\mathbf{W}_P$. □

While Gelfond and Lifschitz consider that only programs with a unique stable model have a well-defined semantics, we take the slightly more general view in this chapter that the semantics of a program is given by the *set* of its stable models.

Definition 8.2.2: We say a sentence S is true according to the stable model semantics of a normal program P if S is true in all stable models of P. Similarly S is false according to the stable model semantics of P if S is false in all stable models of P. □

8.2.2 HiLog Programs

As discussed in Chapter 3, the well founded semantics is applied to normal programs P as follows: all rules are fully instantiated with respect to the Herbrand universe and then the least fixpoint of a corresponding monotonic operator $\mathbf{W}_P$ is taken. Similarly, the stable model semantics can be obtained by first instantiating the program and then identifying 2-valued fixpoints of $\mathbf{W}_P$.

These semantics can be extended to HiLog programs by extending our notion of rule instantiation. Recall that in HiLog, there is no distinction between predicate names and function or constant symbols, so that the Herbrand universe and Herbrand base coincide. We can instantiate the program with respect to this larger Herbrand universe, apply the respective transformations of the well founded or stable model semantics to the instantiated program, and obtain a semantics for HiLog programs.

The question that immediately arises is the following: Does this construction reduce to the standard well-founded semantics (or stable model semantics) for normal logic programs? In its present form the answer is "No."

Example 8.2.1: Consider the normal logic program

$$\begin{array}{l} p \leftarrow \neg q(X) \\ q(a) \end{array}$$

The normal Herbrand universe is the singleton set $\{a\}$, and so the only possible substitution for X in the first rule is $X|a$. Hence p is false with respect to the "normal" well-founded semantics.[4]

However, in the HiLog case, there are other substitutions, such as $X|p$ or $X|a(a,p)$. In this case there *is* an X such that $\neg q(X)$ holds, and so p is true in the HiLog version of the well-founded semantics.

[4] This program behaves anomalously in the normal case due to the lack of extra constants. Adding an unrelated fact such as $r(b)$ would change the answer to the query $?-p$ by supplying an additional constant to the vocabulary. This issue is related to the universal query problem discussed in [85, 118]. One solution is to augment the program with extra function and constant symbols so that this problem does not occur.

Even without negation there can be differences as illustrated by the simple program

$$p(X, X, a)$$

The normal well-founded model is $\{p(a, a, a)\}$. In the HiLog case, the well-founded model is infinite, including $p(X, X, a)$ for X instantiated to each HiLog term involving a and p. $\Box$

The problem with the programs in Example 8.2.1 is that they are not *domain independent*, i.e., the (positive) information provable from the program depends upon the domain for the variables. However, there does exist a large class of normal programs for which the HiLog semantics and the normal semantics coincide, namely the class of range restricted programs. Range restricted programs are guaranteed to be domain independent for normal programs [78].

Recall the definition of range-restrictedness from Chapter 2.

Definition 8.2.3: A normal logic program P is said to be *range restricted* if for every rule r in P, every variable occuring in either the head of r or in a negative literal in the body of r also occurs in a positive literal in the body of r. $\Box$

Before we can prove that the semantics coincide, we need a precise notion of what it means to "coincide." We shall use the notion of conservative extension from Definition 8.1.4.

Theorem 8.2.1: Let P be a range restricted normal logic program. Then the well-founded model of P considered as a HiLog interpretation is a conservative extension of the well-founded model of P considered as a normal interpretation.

Proof: Let P' be the normal instantiation of P, and let Q' be the HiLog instantiation of P. Clearly $P' \subseteq Q'$. Let S be the set of all HiLog atoms from the Herbrand universe that are not normal atoms. We show that S is unfounded with respect to the empty set for the program Q'.

First, observe that no atom in S appears in P'. Let $R = Q' - P'$. Suppose $s \in S$ is the head of an instantiated rule r in R. There is some argument in the head of r that is in S, since $s \in S$ and P is a normal program. This argument must appear in a positive literal in the body by range-restrictedness; let this atom be a. Then a is in S also.

Since the choice of s was arbitrary, we conclude that S is unfounded with respect to the empty set, and so the negation of each member of S will be part of the well-founded model for P considered as a HiLog program.

So the only possibility for new information occurs for instantiated rules r in R with head not in S, and with an atom from S as a subgoal in the body. By range-restrictedness, there must be some positive subgoal with a non-normal argument in r. Since that positive subgoal must be false by the above reasoning, instantiated rules with non-normal terms will not contribute to the well-founded model.

The operator $\mathbf{W}_P$ will then behave similarly to $\mathbf{W}_{P'}$, but making all of S false, and the result is proven. ∎

Theorem 8.2.2: Let P be a range restricted normal logic program. Then there is a one to one correspondence between stable models M' of P considered as HiLog interpretations and stable models M of P considered as normal interpretations, such that M' conservatively extends M.

Proof: Let H be the set of stable models of P considered as a HiLog program, and let N be the set of stable models of P considered as a normal program. Let S be the set of all HiLog atoms from the Herbrand universe that are not normal atoms. By an argument similar to that in Theorem 8.2.1, all atoms in S are false in all stable models in H. By the definition of $\mathbf{W}_P$, if $h \in H$ then the restriction of H to normal atoms generated by P is a stable model in N. Conversely, extending a stable model in N by making all atoms in S false yields a stable model in H. ▮

8.3 Preservation Under Extensions

Informally, a program is domain independent if its semantics does not vary when extra constant symbols are added to the language. To verify domain independence for the well-founded and stable model semantics we add some extra constant, function and predicate symbols to the language to get a larger Herbrand universe, and check that all "newly constructible" atoms are false.

Definition 8.3.1: Let P be a program using a set L of constant, function and predicate symbols, and let L' vary over all supersets of L. We say P is *domain independent* with respect to the well-founded semantics if for every L', the well-founded model M' of P over the larger language L' is a conservative extension of the well-founded model M of P over L. □

We can also formulate a similar definition of domain independence with respect to the stable model semantics.

Definition 8.3.2: Let P be a program using a set L of constant, function and predicate symbols, and let L' vary over all supersets of L. We say P is *domain independent* with respect to the stable model semantics if for every L', there is a surjective map from stable models M' of P over the larger language L' to stable models M of P over L such that M' is a conservative extension of M. □

Example 8.2.1 illustrates some programs that are not domain independent with respect to the well-founded semantics or the stable model semantics. Range restricted normal programs are guaranteed to be domain independent with respect to the well-founded semantics and stable model semantics. (These results are special cases of results proved later in this section.)

We shall be interested in how programs behave under union. We would like the semantics of a program to not be affected by adding extra rules that share no symbols with the original program.

Definition 8.3.3: Let P be a (HiLog) program, and let Q range over all ground programs that have no symbols appearing in P. We say that the well-founded semantics for P is *preserved under extensions* if for all Q the well-founded model of $P \cup Q$ is a conservative extension of the well-founded model of P. □

We can define preservation under extensions similarly for the stable model semantics.

Definition 8.3.4: Let P be a (HiLog) program, and let Q range over all ground programs that have no symbols appearing in P. We say that the stable model semantics for P is *preserved under extensions* if for all Q having at least one stable model there is a surjective map from stable models M of $P \cup Q$ to stable models M' of P such that M is a conservative extension M'. □

Note that for preservation under extenstions with respect to the stable model semantics, we need the extra condition that Q has a stable model. The stable model semantics is not robust under union: for example adding the rule $q \leftarrow \neg q$ to any program destroys all stable models.

For normal programs, preservation under extenstions coincides with domain independence.

Lemma 8.3.1: For both the well-founded semantics and the stable model semantics, a normal program P is domain independent with respect to the semantics if and only if the semantics for P is preserved under extensions.
Proof: (Sketch)

- $\Rightarrow$ If P is domain independent then adding constant symbols does not affect the semantics. Since atoms with different predicate names cannot "interfere" with atoms whose names appear in P, the well-founded semantics and the stable model semantics of P are preserved under extensions.

- $\Leftarrow$ If P is preserved under extensions (with respect to the well-founded or stable model semantics) then we can add a program Q containing an arbitrary set of constants as arguments of predicate names not appearing in P, without affecting the semantics of predicate names appearing in P. Since the semantics of P for predicate names appearing in P is not affected by adding the symbols in Q, P is domain independent (with respect to the semantics). ▮

We have seen how the issue of domain independence arises in normal logic programs, especially in the database context. In HiLog, there is an additional complication due to the presence of variables in predicate names.

Example 8.3.1: Let P be the HiLog program

$$p \leftarrow X(Y), Y(X)$$

for which p is false with respect to the well-founded semantics and stable model semantics. Adding the symbols q and r, together with the facts $q(r)$ and $r(q)$ to the program results in p being true under both the well-founded semantics and the stable model semantics. Hence this program is not preserved under extensions with respect to either semantics. □

The program of Example 8.3.1 is domain independent, since simply adding constants to the language will not change the semantics of p. Hence, for HiLog programs, only half of Lemma 8.3.1 holds: Preservation under extensions implies domain independence, but not the converse. This observation may seem unintuitive since HiLog is supposed to have a "first order semantics." The reason why preservation under extensions is stronger in the HiLog case can be seen when looking at a HiLog program within the universal relation model. Although two programs P and P' may share no symbols, their "universal versions" must share the names of the universal predicates. Hence the HiLog version of preservation under extensions cannot be reduced to the same problem for normal programs.

One may think of domain independence as a *first order* constraint on a program. Preservation under extensions is a *second order* constraint on HiLog programs. Domain independence states, informally, that we don't know what extra values appear in the domain, so we should limit ourselves to the values we know about. For HiLog programs, preservation under extensions states that we don't know what the total set of *predicates* is, so we should limit ourselves to the predicates we know about. Adding other tuples that share no symbols with the original program should not affect the semantics of the original program.

Example 8.3.2: Consider the following transitive closure rules

$$tc(G)(X,Y) \leftarrow G(X,Y)$$
$$tc(G)(X,Y) \leftarrow G(X,Z), tc(G)(Z,Y)$$

This program seems sensible: for every predicate G, $tc(G)$ is the predicate that defines G's transitive closure. However, we must be careful how queries to $tc(G)$ are made.

Suppose the query binds G, so that the query may look like $tc(e_1)(X,Y)$. Then we have no problem; in expanding the right-hand sides of the rules we know which predicates to expand.

However, if the query does not bind G, then while expanding the bodies we must check *every binary predicate name that can be generated by the symbols of the program*, an infinite set of predicates. For example, in the above program we would have to bind G to each of

$$e,\ \ tc(e),\ \ tc(tc(e)),\ \ tc(tc(tc(e))),\ \ldots$$

assuming that e is a symbol appearing in the program.

Further, even though the well-founded semantics of this program is preserved under extensions, the answer set to a query of the form $tc(G)(X,Y)$ (with G unbound) changes if one adds a fact about a binary predicate, even if its symbols are disjoint from those in the original program. For example, adding $salary(john, 30K)$ to the program generates a new answer $tc(salary)(john, 30K)$ to the query. □

Despite the difficulties described in Example 8.3.2, there are ways around the problem. One solution is only to allow references to $tc(G)$ when G is already bound to a ground term, in somewhat similar fashion to the way floundering of negative subgoals is avoided.

An alternative is to bind the predicate name in the rule body itself. In our case, a subgoal $graph(G)$ would restrict G to be a member of the *graph* relation. In either approach, the idea is to restrict the bindings for G to some known set of predicate names.

We take two approaches based on the above observations.

- We generalize range-restrictedness to HiLog programs, and require that queries bind predicate names in the head.
- We consider a subclass of range-restricted HiLog programs that we call strongly range-restricted, for which arbitrary queries are allowed.

In our generalized definition of range-restrictedness the range of *both argument and predicate name variables* is restricted.

Definition 8.3.5: Let r be a HiLog rule. We say r is *range-restricted* precisely when the following conditions hold:

1. Every variable appearing in an argument in the head also appears as an argument in a positive literal in the body.
2. Every variable in a negative literal in the body also appears as an argument in a positive literal in the body, or in the name in the head.
3. There exists an ordering $A_1, \cdots, A_n$ of the positive literals in the body such that if a variable X appears in the predicate name of A_j, then either X appears as an argument in A_k for some $k < j$ or X appears in the predicate name in the head.

If every clause in a HiLog program P is range-restricted, then we say P is range-restricted. We must also restrict the queries that may be asked, so that predicate names are ground in the queries. This condition can be stated as follows: Let $Q(X_1, \ldots, X_n)$ be a query, i.e., a conjunction of literals, where $\{X_1, \ldots, X_n\}$ are all the variables appearing in the query. We say Q is range restricted if the rule

$$answer(X_1, \ldots, X_n) \leftarrow Q(X_1, \ldots, X_n)$$

is range restricted according to the conditions above. □

Definition 8.3.6: Let r be a HiLog rule. We say r is *strongly range-restricted* precisely when the following conditions hold:

1. Every variable appearing in an argument or in the name in the head also appears as an argument in a positive literal in the body.
2. Every variable in a negative literal in the body also appears as an argument in a positive literal in the body.
3. There exists an ordering $A_1, \cdots, A_n$ of the positive literals in the body such that if a variable X appears in the predicate name of A_j, then X appears as an argument in A_k for some $k < j$.

If every clause in a HiLog program P is strongly range-restricted, then we say P is strongly range-restricted. We do not restrict the query for strongly range-restricted programs. □

Note the distinction between variables in predicate *arguments* and variables in predicate *names* in the above definitions.

Example 8.3.3: The following HiLog clauses are strongly range restricted:

$$\begin{array}{l} X(Y)(Z) \leftarrow p(X, Y, W), W(a)(Z), \neg W(b)(Z) \\ p(X) \leftarrow X(a), q(X) \\ tc(G, X, Y) \leftarrow graph(G), G(X, Y) \end{array}$$

The following HiLog clauses are range restricted but not strongly range-restricted:

$$\begin{array}{l} X(Y)(Z) \leftarrow p(Y, Z, W), X(a)(Z), \neg X(b)(Z) \\ tc(G)(X, Y) \leftarrow G(X, Y) \\ not(X)() \leftarrow \neg X \end{array}$$

The following HiLog clauses are not range restricted:

$$\begin{array}{l} X(Y)(Z) \leftarrow Z(X, Y, W), W(a)(Z), \neg W(b)(Z) \\ p(X) \leftarrow X(a) \\ tc(G, X, Y) \leftarrow G(X, Y) \\ not(X) \leftarrow \neg X \end{array}$$

□

Observation 8.3.1: Let P be a range restricted program, Q a ground program, and let $R = P \cup Q$. What can we say about the Herbrand instantiation $H(R)$ of R? We shall be particularly interested in $\mathbf{W}_R = \mathbf{W}_{H(R)}$.

Firstly, $H(R)$ contains $H(Q) = Q$ and $H(P)$. Consider those rules from $H(R)$ that are not in $H(P)$ or Q. We partition these rules into three mutually exclusive categories:

- R_1: Those with all predicate names generated by P, but with some argument not generated by P.
- R_2: Those with some predicate name not generated by P, but with head predicate name generated by P.
- R_3: Those with head predicate name not generated by P.

By range restrictedness, every rule in R_1 has a positive literal in the body that contains an argument not generated by P. Consider rules r in R_2. Some predicate in the body of r must have name not generated by P. Hence this name must appear as an argument of an earlier subgoal. If the earlier subgoal has name not generated by P, then proceed until a subgoal with name generated by P is encountered. Since P is range restricted, we will always reach a subgoal with predicate generated by P but having an argument not generated by P. Using the above observations, it is easy to see that the set of atoms with name generated by P but having some argument not generated by P is unfounded with respect to $\emptyset$, and hence also with respect to every interpretation I. Let $R' = H(P) \cup Q \cup R_3$. It follows then that

$$\mathbf{W}_{H(R)}(I) = \mathbf{W}_{R'}(I)$$

for every interpretation I. Let $\mathbf{W}'_P$ be $\mathbf{W}_P$ but with P instantiated over the Herbrand universe of R rather that that of P. Let $I|_P$ denote the restriction of I to predicate names from P. Then $\mathbf{W}_{R'}(I)$ is a conservative extension of $\mathbf{W}'_P(I)|_P$ for every I, since no rules in Q or R_3 have heads with names generated by P.

For strongly range-restricted programs P, we can make a stronger observation. If P is strongly range-restricted then every rule in R_3 will have a subgoal with name generated by P but argument not generated by P. Since each such atom is in the greatest unfounded set with respect to $\emptyset$, we can conclude that for strongly range-restricted programs that

$$\mathbf{W}_{H(R)}(I) = \mathbf{W}_{R''}(I)$$

for all interpretations I, where $R'' = H(P) \cup Q$. ∎

Lemma 8.3.2: Let P be a range restricted HiLog program, Q a ground HiLog program, and let $R = P \cup Q$. Let M_R be a fixpoint of $\mathbf{W}_R$, and let M_P be M_R restricted to atoms in the Herbrand base of P. Then M_P is a fixpoint of $\mathbf{W}_P$, and M_R is a conservative extension of M_P.
Proof: By Observation 8.3.1 it follows that $\mathbf{W}_R(M_R)$ is a conservative extension of $\mathbf{W}'_P(M_R)|_P$. Further, since every atom with predicate name generated by P but having an argument not generated by P is false in M_R by Observation 8.3.1, $\mathbf{W}'_P(M_R)|_P$ is a conservative extension of $\mathbf{W}_P(M_P)$. Thus $M_R = \mathbf{W}_R(M_R)$ is a conservative extension of $\mathbf{W}_P(M_P)$. It follows that $M_R|_P$ conservatively extends $\mathbf{W}_P(M_P)$, and so M_P must too. Since M_P and $\mathbf{W}_P(M_P)$ have exactly the same constant, function and predicate symbols, by the definition of conservative extension we have $M_P = \mathbf{W}_P(M_P)$.

Finally, M_R is a conservative extension of M_P since, by Observation 8.3.1, every atom with name generated by P but having an argument not generated by P is false in M_R. ∎

Theorem 8.3.3: The well-founded semantics of range restricted HiLog programs is preserved under extensions.

Proof: We need to show that the *least* fixpoint of $\mathbf{W}_R$ from Lemma 8.3.2 above is in fact a conservative extension of the *least* fixpoint of $\mathbf{W}_P$. Suppose the contrary, i.e., that M_R was the least fixpoint of $\mathbf{W}_R$, but that M_P was not the least fixpoint of $\mathbf{W}_P$. Let M'_P be a smaller fixpoint of $\mathbf{W}_P$, and let M''_P be the conservative extension of M'_P over the Herbrand base of R. Iterate $\mathbf{W}_R$ starting with M''_P until a fixpoint M'_R is reached. $M'_R \subseteq M_R$ by the monotonicity of $\mathbf{W}_R$, since $M''_P \subseteq M_R$. Iterating $\mathbf{W}_R$ from M''_P cannot generate new literals from P since, as in Observation 8.3.1, $\mathbf{W}_R(I)$ is a conservative extension of $\mathbf{W}'_P(I)|_P$ for every I. Hence M'_R is a proper subset of M_R since there must be some literal from P in M_P (and hence in M_R) but not in M''_P (or M'_R). Thus we contradict the assumption that M_R was the least fixpoint of $\mathbf{W}_R$. ▮

Theorem 8.3.4: The stable model semantics of strongly range restricted HiLog programs is preserved under extensions.

Proof: We need to show that every stable model of P is conservatively extended by a stable model of R. Let M_P be an arbitrary stable model of P, and let M_Q be a stable model of Q. Let M_R be $M_P \cup M_Q$ (over the larger universe, making false all atoms not generated by P alone or Q alone). By Observation 8.3.1 on strongly range-restricted programs, $\mathbf{W}_{H(P\cup Q)} = \mathbf{W}_{H(P)\cup Q}$. Since $H(P)$ and Q share no symbols, it follows that M_R is a stable model of $P \cup Q$, being composed of stable models of P and Q. M_P is the restriction of M_R to literals generated by P, so that M_R does conservatively extend M_P. Hence some stable model of $P \cup Q$ conservatively extends M_P. Since M_P was chosen arbitrarily, we are done. ▮

Unlike the case for the well-founded semantics, Theorem 8.3.4 does not hold for some programs that are range restricted but not strongly range restricted. For example if P is the range-restricted program

$$X(a) \leftarrow X(X), \neg X(a)$$

and Q contains the single fact $r(r)$, then $P \cup Q$ has no stable model even though each of P and Q have stable models.

Since domain independence is undecidable for normal programs [31] it follows that preservation under extensions for HiLog programs is also undecidable. Thus we must resign ourselves to sufficient conditions such as range restrictedness.

8.4 Modular Stratification

In general, the well-founded semantics and the stable model semantics are 3-valued, in the sense that an atom may be true, false, or undefined. There are many situations in which two truth values, namely true and false, suffice. For normal programs the class of stratified programs is a well-known class of programs with a two-valued semantics. Stratification is syntactically recognisable. (See Definition 2.1.9 and discussion in Section 3.1.) Stratification can be generalized to "local stratification" (Definition 2.1.10).

In Section 5.2 we defined the class of "modularly stratified programs," and have shown that the well-founded semantics (and hence also the stable model semantics) are two-valued for this class of normal programs. Modular stratification was used in Chapter 5 as a basis for defining a magic-sets transformation to evaluate normal programs with negation.

What can be said about various forms of stratification in HiLog? Is there a reasonable analog of these concepts when predicates can be dynamic, instantiated by other subgoals? It is tempting to say that HiLog is "first order," and to use the universal relation model to reduce the problem to the problem for normal programs.

Unfortunately, the transformation to universal relations obscures the structure of the program. A normal stratified program may not be stratified under the universal relation model since we replace predicate names by generic relations of the appropriate arity, so that the stratified program

$$p(X) \leftarrow q(X), \neg r(X)$$

becomes unstratified, namely

$$u_2(p, X) \leftarrow u_2(q, X), \neg u_2(r, X).$$

Similarly, strongly connected components used in the definition of modular stratification are merged under the universal relation model, and so the level at which predicates are used is lost in the universal version of the program. In this section we ignore the universal relation model, and look for conditions on the original program that ensure a two-valued semantics.

If all predicate names are variable-free in the program, then we may generalize each of the above forms of stratification in the obvious way. However unless variables *do* appear in predicate names there is no point using HiLog rather than normal programs. Having nonground terms in predicate names is *the main advantage* of HiLog over normal programs.

We can require only that the outermost functor of every predicate name is ground, and assign levels to the outermost functor (or for local or modular stratification, to instances of atoms with that outermost functor).[5] This proposal is more general than eliminating variables from predicate names altogether, and the version of modular stratification that results can capture the game program of Example 5.2.1.

However, it is unclear how to generalize these concepts when variables appear as the outermost predicate constructor. For example, we may wish to generalize Example 5.2.1 to make the particular game under consideration a parameter, so that we don't need a separate set of "winning" rules for each game. Such a generalized rule might look like

$$winning(M)(X) \leftarrow game(M), M(X, Y), \neg winning(M)(Y)$$

which has a variable as a predicate name in the body.

Recall that to define modular stratification we first divided the program into its mutually recursive components. Unfortunately, we cannot divide an arbitrary range restricted program into its mutually recursive components a priori, since the components may vary depending on values of other predicates.

Example 8.4.1: Consider the (strongly) range restricted program

$$X(a, b) \leftarrow p(X, Y), \neg Y(a, b)$$

together with some tuples for the predicate p. By including appropriate tuples in p, one can see that *any* set of mutually recursive components can be specified. Further, any partial ordering on these components can be specified. For example, if we want q_1, q_2 and q_3 to be in the same

[5]It is possible to assign different levels to occurrences of the same predicate with different arities.

component, r and s to each be in its own component, and for r to have a higher component than s, we could include the tuples

$$p(q_1, q_2), p(q_2, q_1), p(q_1, q_3), p(q_3, q_1), p(r, s)$$

in the extension of p. We could even put tuples like $p(p, q)$ and $p(q, p)$ in the program to force p itself to be in the same component as any other predicate. □

Nevertheless, all is not lost. We can generalize modular stratification to (strongly range-resticted) HiLog programs by identifying the lowest components one at a time, rather than all components at once. We present a procedure below and define a program to be modularly stratified for HiLog if the procedure says so. The generalization of reduction (Definition 5.2.3) to HiLog rules is defined below.

Definition 8.4.1: (HiLog reduction) Let F be a collection of rules, and let S be a set of "settled" predicates. Suppose S is fully defined by an interpretation M over the universe $\mathcal{U}$, with terms interpreted freely.

Form $I_{\mathcal{U},S}(F)$, the (partial) instantiation of F with respect to $\mathcal{U}$ and S, by substituting terms from $\mathcal{U}$ in every possible way for all variables in the arguments of literals appearing in the rules of F with names from S. Delete from $I_{\mathcal{U},S}(F)$ all rules having a subgoal whose predicate is in S, but which is false in M. From the remaining rules, delete all subgoals having predicates in S (which must be true) to leave a set of partially instantiated rules $R_M(F)$. We call $R_M(F)$ the *HiLog reduction of F modulo M*. □

We can only assume that the interpretation modulo which we are reducing the program is for lower components. (If it turns out that a predicate name that should already have been fully defined shows up in the head of a reduced rule, then we'll say that the program is not modularly stratified.) Also, variables in predicate names are instantiated only through the instantiation of that variable appearing as an argument elsewhere in the rule. Atoms with variables in the predicate name cannot be instantiated until the predicate name becomes ground.

Definition 8.4.2: (Modular Stratification for HiLog) Let P be a strongly range-restricted HiLog program. We say P is modularly stratified if the procedure of Figure 8.1 terminates and outputs that P is modularly stratified. R denotes the set of rules still to process, S denotes all those predicate names that are already settled, and M denotes the partially computed well-founded model for predicates in S. □

Example 8.4.2: Consider the game program P consisting of the rule

$$winning(M)(X) \leftarrow game(M), M(X,Y), \neg winning(M)(Y)$$

together with the tuples $game(move_1)$, $game(move_2)$, and tuples for the acyclic relations $move_1$ and $move_2$. Using the procedure of Definition 8.4.2 we would first partition P into the first rule and the remainder. The remainder, being a set of facts, is locally stratified, and thus has a two-valued well-founded model in which only the explicitly listed facts about $game$, $move_1$ and $move_2$ are true.

This model is used to reduce the first rule, giving the rule

$$winning(move_1)(X) \leftarrow move_1(X,Y), \neg winning(move_1)(Y)$$

```
Initialize R to P, S to ∅, and M to ∅.

While R is nonempty do {

Partition R into R1 and R2, where R1 contains all rules with variables in
head predicate names, and R2 contains the remaining rules.  If R2
contains any rules with a head predicate from S or if R2 is empty then
exit('P is not modularly stratified for HiLog').

Construct a directed graph G as follows:  The nodes of G are all
predicate names appearing ground in R.  For every rule r in R2, if p is
the head predicate name of r and q is a ground predicate name in the body
of r then construct an edge from p to q.

Find the strongly connected components of G, and let T be the set of
nodes in G from components with no outgoing edge.

Let RT be the subset of R2 with head predicate names from T.  If RT is
not locally stratified or if RT contains a rule with a variable in a
predicate name then exit('P is not modularly stratified for HiLog').

Let MT be the well-founded model for RT.  S := S ∪ T, M := M ∪ MT, and
replace R with the HiLog reduction of R − RT modulo M.  }

exit('P is modularly stratified for HiLog').
```

Figure 8.1: Procedure for determining modular stratification.

and similarly for $move_2$. These rules can be further reduced now that the variable M has been instantiated, giving instances of

$$winning(move_1)(X) \leftarrow \neg winning(move_1)(Y)$$

when $move_1(X, Y)$ holds, and similarly for $move_2$. These rules form a locally stratified program since the two move relations are acyclic, and so the program is modularly stratified for HiLog. □

Note that if we did not require the program to be *strongly* range restricted, then there would be no guarantee that predicate names in the head eventually become ground. In any case, having variables in predicate names in the head is a potential source of violations of modular stratification. Problems can occur when the predicate in the head is used in the body of another rule.

Example 8.4.3: Consider a version of Example 8.4.2 in which $move_1$ is not an EDB predicate, but an IDB predicate. Suppose we include the rules

$$\begin{array}{l} X \leftarrow p(X) \\ p(X) \leftarrow q(X) \end{array}$$

and include the original $move_1(X, Y)$ tuples as tuples of the form $q(move_1(X, Y))$. Then the rules above seem to generate the $move_1$ tuples needed to get modular stratification of the whole program.

However, what happens is that after the first reduction, we get the rule

$$winning(move_1)(X) \leftarrow move_1(X, Y), \neg winning(move_1)(Y)$$

and then try to reduce this rule using the well-founded model for the rules with head $move_1$. The problem is that there are no such rules, since the rule $X \leftarrow p(X)$ has not itself been reduced yet. Hence, the well-founded model will make $move_1$ universally false. When the rule $X \leftarrow p(X)$ does get instantiated with X bound to $move_1$ we violate the condition that rule heads not contain predicates already settled, and so the program is not modularly stratified for HiLog. □

While the program of Example 8.4.3 may seem innocuous, it may be possible that some other rule for p depends negatively on *winning*, say. Further, we may not be able to recognize such a dependency just by looking at the program since p may refer to a negative subgoal with a variable as the predicate name. Our approach is a conservative one: assume that variables in head predicate names don't conflict with the current lowest component, and if a conflict appears later then reject the program.

On the other hand, some programs that appear problematic are in fact modularly stratified for HiLog. Consider the program

$$Q(a) \leftarrow p(Q), \neg p(a).$$

Instantiating Q to p would yield a program that is not locally stratified. However, the program *is* modularly stratified for HiLog, since the well-founded model M for rules with head p is computed first. Since there don't happen to be any such rules, p is universally false according to M. Hence the reduction of this rule modulo M is empty, and thus the program is modularly stratified for HiLog.

In Definition 8.4.2 above, we have actually constructed the well-founded model, which is the union of the well-founded models at each step. This observation leads to the following result.

Theorem 8.4.1: If a program is modularly stratified for HiLog then it has a total well-founded model that is its unique stable model.

Proof: During the execution of the procedure of Figure 8.1 we effectively compute the well-founded model. The critical observation is that when constructing M_T, we can see *all* (instantiated) rules with heads having predicate names for T. If, at some later stage a head is instantiated to a name from T, then the program is not modularly stratified for HiLog, and there is nothing to prove. ▮

We must also satisfy ourselves that modular stratification for HiLog generalizes modular stratification for normal programs.

Lemma 8.4.2: Let P be a normal program. P is modularly stratified for HiLog if and only if it is modularly stratified.

Proof: By specializing the procedure of Figure 8.1 to normal programs, we can observe that on each round the lowest remaining strongly connected components are selected and their union is tested for local stratifiability. Since all predicate names are ground, there will never be conflicts with predicates already in S, and so the procedure of Figure 8.1 reduces to an iterative implementation of Definition 5.2.4. ▮

While in general one needs to go through the construction above to decide whether a program is modularly stratified, there are many cases where knowledge about acyclicity of certain predicates can give a syntactic check for modular stratification. For example, in the game program of Example 8.4.2 the constraints needed are that *game* is at a level no higher than all its arguments, and that any argument of *game* is an acyclic binary relation. If *game* is at no higher level that its arguments then it will not be in a higher component, and will have its semantics computed "in time." For each e such that $game(e)$ holds, we will effectively be left with an instantiated rule

$$winning(e)(X) \leftarrow \; e(X,Y), \neg winning(e)(Y).$$

The remaining program is modularly stratified, as before, if each such e is acyclic.

As in Section 5.6, one can extend the idea of modular stratification to operators such as aggregation that have traditionally been stratified to avoid semantic difficulties. For example, suppose we have relations $part_i(X,Y,N)$ that are true when X has N copies of Y as an immediate subpart in machine number i.[6] We can solve the parts-explosion problem for aritrary part-subpart relations using the following HiLog program.

$$\begin{aligned}
&in(Mach,X,Y,null,N) \leftarrow assoc(Mach,Part), Part(X,Y,N)\\
&in(Mach,X,Y,Z,N) \leftarrow assoc(Mach,Part), Part(X,Z,P), contains(Mach,Z,Y,M),\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad N = P * M\\
&contains(Mach,X,Y,N) \leftarrow N = \texttt{sum}\ P : in(Mach,X,Y,_,P)
\end{aligned}$$

(The sum in the third rule is grouped by *Mach*, X and Y; for each *Mach*, X and Y we sum all corresponding P.) The *assoc* relation stores with each machine name m_i the name of the part relation $part_i$ that defines the machine.[7] The `sum` operation here is not stratified. *contains*

[6] We adopt the convention that we are only interested in smaller, simpler subparts.

[7] We prefer to use an *assoc* relation rather than adding *Mach* as an argument to a generic *part* relation. Having an *assoc* relation allows machines that share part-hierarchies, and machines that consist of the union of several part-hierarchies, without having to represent the hierarchies themselves more than once. It also avoids an extra argument in every *part* tuple.

depends on itself through aggregation, via the predicate *in*. However, assuming each $part_i$ is acyclic in its first two arguments, the summation operates on successively lower arguments (i.e., smaller subparts), and so there is no looping through summation. This is the aggregate analog of modular stratification. By using HiLog rather than normal programs, we can write a solution to the parts-explosion problem once, rather than once for each particular part-subpart relation.

8.4.1 Magic Sets

In Chapter 5, modular stratification was used as a basis for defining a magic sets transformation for range-restricted programs. We can use the same observations to define a similar magic sets transformation for strongly range-restricted HiLog programs that are modularly stratified for HiLog. We do not present a full description of the method. We describe only the additional features required in the HiLog case.

Firstly, we require that a program satisfies an analog of modular stratification from left to right (Definition 5.2.6). The appropriate refinement of modular stratification for HiLog is in the construction of the graph G described in Figure 8.1. Rather than constructing edges from the head predicate to *all* ground predicates in the body, we construct edges only to the leftmost predicate in the body. We assume that the subgoals of rules are ordered so that the corresponding refined version of modular stratification holds, and also so that the program does not flounder.[8]

Since the program is assumed to be strongly range-restricted, queries with variables in their names are permitted. If Q is an atom and ?-Q is the query, then the tuple $magic(Q, +)$ appears in the rewritten program. As far as supplementary predicates are concerned, variables in names and variables in arguments are treated the same. One technical difference is that because of variables in subgoal names, it may not be clear which are EDB predicates and which are IDB predicates. As a result, we have to assume (unless further information is given) that all predicates are IDB predicates. The remainder of the transformation is essentially unchanged, except that literals are HiLog literals, and may have variables in their names.

Example 8.4.4: The (abbreviated) game program

$$w(M)(X) \leftarrow g(M), M(X,Y), \neg w(M)(Y)$$

with the query ?-$w(m)(a)$ would be rewritten to yield the rules in Figure 8.2. These rules, together with some rewritten rules for m and g, can be executed bottom-up (in the order described in Section 5.4.4). □

While the magic transformation of Chapter 5 is well-defined for all range-restricted, modularly stratified normal programs, the evaluation of the transformed program is not guaranteed to terminate for programs with recursively applied function symbols. For Datalog programs, though, termination is guaranteed.

In a similar way, the corresponding magic sets transformation for HiLog programs may not terminate if there are recursively applied function or predicate symbols. Nevertheless, we can guarantee termination in a case corresponding to Datalog, where symbols are never used as functions, and where the only use of variables in predicate names is when the predicate name is a single variable.

[8] A HiLog program flounders if a negative subgoal containing variables is leftmost, or if a subgoal with a variable in its name is leftmost.

$$magic(w(m)(a), +)$$

$$
\begin{aligned}
&sup_{1.0}(M, X) \leftarrow magic(w(M)(X), \_) \\
&w(M)(X) \leftarrow sup_{1.3}(M, X) \\
&magic(M(X, Y), +) \leftarrow sup_{1.1}(M, X) \\
&magic(w(M)(Y), -) \leftarrow sup_{1.2}(M, X, Y) \\
&sup_{1.1}(M, X) \leftarrow sup_{1.0}(M, X), g(M) \\
&sup_{1.2}(M, X, Y) \leftarrow sup_{1.1}(M, X), M(X, Y) \\
&sup_{1.3}(M, X) \leftarrow sup_{1.2}(M, X, Y), \Box w(M)(Y) \\
&dp(w(M)(X), M(X, Y)) \leftarrow magic(w(M)(X), -), sup_{1.1}(M, X) \\
&dp(P, M(X, Y)) \leftarrow dp(P, w(M)(X)), sup_{1.1}(M, X) \\
&dn(w(M)(X), w(M)(Y)) \leftarrow magic(w(M)(X), -), sup_{1.2}(M, X, Y) \\
&dn(P, w(M)(Y)) \leftarrow dp(P, w(M)(X)), sup_{1.2}(M, X, Y) \\
&dn'(Q) \leftarrow magic(Q, -), Q \\
&dn'(Q) \leftarrow magic(Q, -), \Box Q
\end{aligned}
$$

$$\Box P \leftarrow magic(P, -), \forall Q(dn(P, Q) \Rightarrow dn'(Q)), \sim P$$

Figure 8.2: Rewritten rules from Example 8.4.4.

Definition 8.4.3: We say that a HiLog program P is a *Datahilog* program if for every atom in a rule of P, both the name and arguments of the atom are either variables or constant symbols. $\Box$

The rule

$$winning(M, X) \leftarrow game(M), M(X, Y), \neg winning(M, Y)$$

is a Datahilog program, while

$$tc(G)(X, Y) \leftarrow graph(G), G(X, Z), tc(G)(Z, Y)$$

is not.

Lemma 8.4.3: The set of ground atoms that are not made false according to the well-founded semantics of a strongly range-restricted Datahilog program P is finite.

Proof: Let H be the Herbrand base of P, let A be the set of all arities appearing in P, and let T be the set of terms of the form $c_0(c_1, \ldots, c_n)$, where each c_i is a constant symbol appearing in P and $n \in A$. Clearly T is finite. We shall show that $H - T$ is an unfounded set with respect to $\emptyset$, and hence are false according to the well-founded semantics. Thus the true and undefined atoms are all in T and hence form a finite set.

Consider an arbitrary element h of $H - T$. Either the name of h or one of its arguments is a complex term. In either case, that complex term must appear in an argument of a positive literal in the body of any instantiated rule with head h. That positive literal is in $H - T$; since this argument holds for all rules in P, we conclude that $H - T$ is unfounded with respect to $\emptyset$. ∎

Lemma 8.4.3 does not hold for range-restricted programs that are not strongly range-restricted, as illustrated by the simple program

$$X(a, b).$$

On the other hand, the extent of any given (ground) predicate is finite in a range-restricted program, by an argument similar to that in Lemma 8.4.3.

Chapter 9

Conclusions

9.1 Our Contribution

The work in this thesis contributes in two main areas: semantics for deductive databases, and evaluation techniques for deductive databases.

We have proposed the well-founded semantics for deductive databases (or logic programs) with negation. This semantics generalizes a number of previous approaches, while being well-defined for all programs.

We have provided a top-down evaluation procedure called global SLS-resolution that is sound and complete with respect to the well-founded semantics.

We have isolated the class of modularly stratified programs, and shown how to extend magic-sets techniques to this class of programs. Previous approaches only handled stratified programs. We have demonstrated a one-to-one correspondence between steps of the bottom-up magic-sets evaluation and a generalized top-down method with memoing.

We have shown how to augment magic-sets techniques to incorporate an optimization by tail-recursion elimination. This optimization provides a guaranteed speedup for a relatively large class of programs, significantly generalizing the techniques of right-linear evaluation.

We have generalized the class of acyclic programs to the class of modularly acyclic programs, and generalized some semantic and computational results from acyclic programs to modularly acyclic programs.

Finally, we have considered the second order logic HiLog, and shown how to define the well-founded and stable model semantics for HiLog programs. We have identified the property of "preservation under extensions" which is a second order analog of domain independence. We have generalized modular stratification to HiLog programs.

9.2 Further Work

9.2.1 Arithmetic and Aggregation

In realistic database systems, operations such as arithmetic and aggregation are mandatory. In particular, such operations should be allowed in a declarative language. However, certain features such as termination of bottom-up evaluation, cannot be guaranteed in the presence of arithmetic, as illustrated by the program

$$p(X) \leftarrow p(Y), X = Y + 1$$
$$p(0)$$

A semantics for arithmetic operations can be defined simply by incorporating distinguished arithmetic functions (with appropriate behaviors) in the formal model. Providing a sound semantic basis for declarative systems with aggregation is an important issue. Recursion through aggregation causes similar semantic problems to recursion through negation. Some results in this direction have been reported in [35, 50, 99, 110].

9.2.2 Disjunctive Programs

In [95] we generalize the well-founded semantics to disjunctive programs. Disjunctive programs allow information of the form $p \vee q$ without knowing p or q. We generalized the semantics in two ways, one "exclusive" in nature and the other "inclusive" in nature. There have been a number of other proposals for semantics in disjunctive databases [22, 61, 70, 82, 87, 100, 102].

There remain several problematic aspects to disjunctive reasoning. One issue is complexity. With some exceptions, authors have ignored the complexity of their semantics. In most cases, the semantics are likely to be NP-hard even for propositional programs, since the reasoning mechanisms must test some version of (at least) propositional satisfiability.

In light of the complexity argument above, one must ask in what circumstances is disjunctive reasoning absolutely necessary in a deductive database system. Even if necessary, one may choose to avoid the complexity of general disjunctive reasoning by isolating the disjunctive information to a small component of the database.

Another issue is the level of knowledge required to understand the various proposals for disjunctive programs. For some semantics, given a disjunctive program it is often difficult (without manually computing the semantics) to decide whether a sentence is implied by the semantics.

9.2.3 Explicit Negation

The programs we have considered in this thesis do not allow the specification of explicit negative information. As in relational algebra or calculus, one assumes that one has complete information, and that an atom is false unless it is derivable in some sense.

Several authors have considered adding explicit negation to programs [38]. The idea is to allow explicit negative information to be represented in addition to information inferred false by default. The issue of complexity arises once more, as one generally needs to perform consistency checks.

9.2.4 Constructive Negation

Suppose one has a program

$$p(X) \leftarrow \neg q(X)$$
$$q(a)$$

If we were to ask the query ?-$p(X)$ on this program a method such as SLDNF-resolution or global SLS-resolution would flounder, trying to expand the negative literal $\neg q(X)$ containing a variable. One may like to give the "answer" $X \neq a$. However the traditional definition of answer substitution does not permit such "nonsubstitutions." Several authors have considered generalizing the notion of answer substitution to incorporate inequality substitutions [21, 84]. For function-free programs there is a normal form for such substitutions.

In it unclear whether there is an overwhelming need for constructive negation in deductive databases; such a need would have to be demonstrated by empirical evidence. There is scope to generalize bottom-up techniques to handle constructive negation. However, using constructive negation may add to the cost of some fundamental operations such as composing substitutions.

9.2.5 Prototype Systems

One of the most important issues for deductive databases is designing and using deductive database systems. Deductive database theory is reasonably mature, having been studied in its own right for almost a decade, and, in the context of logic programming, for longer. However, deductive database systems have only recently become available. For descriptions of various systems see [75, 79, 80, 89, 114].

The main advantages of deductive databases are expressiveness and declarativeness. One needs to be convinced that the extra expressiveness gives a *useful* extension to, say, relational algebra. Some detractors have claimed that "recursion is never used in practice." It may be argued that recursion is not used in practice now because it is not available in a declarative way. Nevertheless, the onus is on deductive database system designers to demonstrate applications that use features such as recursion.

Complex applications, such as computer-aided design, various engineering problems, and financial planning applications, are good candidates for deductive database use. Demonstrating such applications is critical for the successful adoption of the deductive database paradigm.

9.2.6 Object Oriented Deductive Databases

Object-oriented databases have become popular in recent years due to their emphasis of such features as encapsulation, complex objects and object identity. However the basis for such systems is rarely a declarative one.

In languages such as HiLog it is possible to define parameterized procedures, or in the object-oriented terminology, "methods." A promising approach is the integration of some object oriented features in a deductive database framework. Various such attempts have been reported in [2, 44, 62, 123]. A successful integration would provide mechanisms for encapsulation, inheritance, complex objects and other object-oriented concepts while retaining the declarativeness and formal basis of deductive databases.

Bibliography

[1] H. Abelson and G. J. Sussman. *Structure and Interpretation of Computer Programs*. McGraw Hill, 1985.

[2] S. Abiteboul. Towards a deductive object-oriented database language. In *Proceedings of the First International Conference on Deductive and Object-Oriented Databases*, 1989.

[3] S. Abiteboul and V. Vianu. Procedural and declarative database update languages. In *ACM Symposium on Principles of Database Systems*, 1988.

[4] A. V. Aho and J. D. Ullman. Universality of data retrieval languages. In *6th ACM Symp. on Principles of Programming Languages*, pages 110–120, 1979.

[5] K. R. Apt and M. Bezem. Acyclic programs. In *Proceedings of the Seventh International Conference on Logic Programming*, 1990.

[6] K. R. 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, Los Altos, CA, 1988. Morgan Kaufmann.

[7] K. R. Apt and M. H. Van Emden. Contributions to the theory of logic programming. *JACM*, 29(3):841–862, 1982.

[8] I. Balbin, K. Meenkashi, and K. Ramamohanarao. An efficient labelling algorithm for magic set computation on stratified databases. Technical Report 88/1, Dept. of Computer Science, University of Melbourne, 1988.

[9] I. Balbin, G. S. Port, and K. Ramamohanarao. Magic set computation for on stratified databases. Technical Report 87/3, Dept. of Computer Science, University of Melbourne, 1987.

[10] I. Balbin, G. S. Port, K. Ramamohanarao, and K. Meenkashi. Efficient bottom-up computation of queries on stratified databases. *Journal of Logic Programming*, 1989. (to appear).

[11] F. Bancilhon, D. Maier, Y. Sagiv, and J. D. Ullman. Magic sets and other strange ways to implement logic programs. In *Proceedings of the Fifth ACM Symposium on Principles of Database Systems*, 1986.

[12] F. Bancilhon and R. Ramakrishnan. An amateur's introduction to recursive query processing strategies. In *1986 ACM-SIGMOD Conf. on Management of Data*, pages 16–52, 1986.

[13] C. Beeri and R. Ramakrishnan. On the power of magic. *Journal of Logic Programming*, 10:255–300, 1991. Preliminary version appeared in the 6th ACM Symposium on Principles of Database Systems, 1987.

[14] C. Beeri, R. Ramakrishnan, D. Srivastava, and S. Sudarshan. Magic implementation of stratified logic programs. (manuscript), 1989.

[15] A. Brodsky and Y. Sagiv. Inference of monotonicity constraints in Datalog programs. In *Proceedings of the Eighth ACM Symposium on Principles of Database Systems*, 1989.

[16] L. Brownston. *Programming Expert Systems in OPS5: An Introduction to Rule-based Programming.* Addison Wesley, 1985.

[17] F. Bry, Dec. 1989. (personal communication).

[18] F. Bry. Logic programming as constructivism: A formalization and its application to databases. In *Proceedings of the Eighth ACM Symposium on Principles of Database Systems*, 1989.

[19] F. Bry. Query evaluaton in recursive databases: Bottom-up and top-down reconciled. In *Proceedings of the First International Conference on Deductive and Object-Oriented Databases*, 1989.

[20] L. Cavedon. Continuity, consistency, and completeness properties for logic programs. In *Proceedings of the Sixth International Conference on Logic Programming*, 1989.

[21] D. Chan. Constructive negation based on the completed database. In *Proc. Fifth International Conference and Symposium on Logic Programming*, 1988.

[22] E. P. F. Chan. A possible world semantics for non-Horn databases. Technical Report CS-89-47, University of Waterloo, 1989.

[23] A. Chandra and D. Harel. Structure and complexity of relational queries. *JCSS*, 25(1):99–128, 1982.

[24] A. Chandra and D. Harel. Horn clause queries and generalizations. *Journal of Logic Programming*, 2(1):1–15, 1985.

[25] W. Chen, M. Kifer, and D. S. Warren. HiLog: A first order semantics for higher-order logic programming constructs. In *Proc. North American Logic Programming Conference*, 1989.

[26] P. Cholak. Post correspondence problem and Prolog programs. manuscript, Dept. of Mathematics, University of Wisconsin, 1988.

[27] K. L. Clark. Negation as failure. In Gallaire and Minker, editors, *Logic and Databases*, pages 293–322. Plenum Press, New York, 1978.

[28] K. L. Clark. Predicate logic as a computational formalism. Technical Report 79/59, Department of Computing, Imperial College, 1979.

[29] T. Cooper. *Rule-based Programming with OPS5.* Morgan Kaufmann, 1988.

[30] S. Dietrich and D. S. Warren. Dynamic programming strategies for the evaluation of recursive queries. Technical Report 85/31, Computer Science Department, State University of New York at Stony Brook, 1985.

[31] R. A. DiPaola. The recursive unsolvability of the decision problem for the class of definite formulas. *JACM*, 16(2):p.324, 1969.

[32] M. A. E. Dummett. *Elements of Intuitionism.* Clarendon Press, Oxford, 1977.

[33] Ph. M. Dung and K. Kanchanasut. A natural semantics for logic programs with negation. Technical report, Asian Institute of Technology, Bankok 10501, Thailand, 1989. (manuscript).

[34] M. Fitting. A Kripke-Kleene semantics for logic programs. *Journal of Logic Programming*, 2(4):295–312, 1985.

[35] S. Ganguly, S. Greco, and C. Zaniolo. Minimum and maximum predicates in deductive databases. In *Proceedings of the Tenth ACM Symposium on Principles of Database Systems*, 1991.

[36] M. Gelfond. On stratified autoepistemic theories. In *Proc. AAAI*, 1987.

[37] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proc. Fifth International Conference and Symposium on Logic Programming*, 1988.

[38] M. Gelfond and V. Lifschitz. Logic programs with classical negation. manuscript, September 1989.

[39] M. R. Genesereth and M. L. Ginsberg. Logic programming. *C. ACM*, 28:933–941, 1985.

[40] M. R. Genesereth and N. J. Nilsson. *Logical Foundations of Artificial Intelligence.* Morgan Kaufman, 1987.

[41] S. A. Greibach. A new normal form theorem for context-free phrase structure grammars. *J.ACM*, 12(1):42–52, 1965.

[42] Y. Gurevich and S. Shelah. Fixed-point extensions of first order logic. *Annals of Pure and Applied Logic*, 32:265–280, 1986.

[43] S. Hanks and D. McDermott. Default reasoning, nonmonotonic logics, and the frame problem. In *AAAI Conference*, pages 328–333, 1986.

[44] A. Heuer and P. Sander. Semantics and evaluation of rules over complex objects. In *Proceedings of the First International Conference on Deductive and Object-Oriented Databases*, 1989.

[45] R. Hill. Lush resolution and its completeness. Technical Report DCL Memo 78, Department of Artificial Intelligence, University of Edinburgh, 1974.

[46] J. E. Hopcroft and J. D. Ullman. *Introduction to automata theory, languages, and computation.* Addison-Wesley, Reading, MA, 1979.

[47] N. Immerman. Relational queries computable in polynomial time. *Information and Control*, 68(1):86–104, 1986.

[48] J. Jaffar, J.-L. Lassez, and J. Lloyd. Completeness of the negation-as-failure rule. In *Int'l Joint Conf. on Artificial Intelligence*, pages 500–506, 1983.

[49] D. B. Kemp, K. Ramamohanarao, and Z. Somogyi. Right-, left- and multi-linear rule transformations that maintain context information. In *Proceedings of the International Conference on Very Large Databases*, 1990.

[50] D. B. Kemp and P. J. Stuckey. Semantics of logic programs with aggregates. In *Proceedings of the International Logic Programming Symposium*, 1991.

[51] D. B. Kemp and R. W. Topor. Completeness of a top down query evaluation procedure for stratified databases. In *Proc. Fifth International Conference and Symposium on Logic Programming*, 1988.

[52] J. M. Kerisit and J. M. Pugin. Efficient query answering on stratified databases. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, 1988.

[53] P. G. Kolaitis. The expressive power of stratified programs. *Information and Computation*, 90:50–66, 1991.

[54] P. G. Kolaitis and C. H. Papadimitriou. Why not negation by fixpoint? In *ACM Symposium on Principles of Database Systems*, 1988.

[55] R. A. Kowalski. Algorithm = Logic + Control. *C.ACM*, 22(7):424–436, 1979.

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

[57] K. Kunen. Some remarks on the completed database. Technical Report 775, Univ. of Wisconsin, Madison, WI 53706, 1988. (Abstract appeared in 5th Int'l Conf. Symp. on Logic Programming, Seattle, Aug. 1988).

[58] J.-L. Lassez and K. Marriott. Explicit representation of terms defined by counter examples. *Journal of Automated Reasoning*, 3:301–317, 1987.

[59] 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, Los Altos, CA, 1988. Morgan Kaufmann.

[60] J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, New York, 2nd edition, 1987.

[61] J. Lobo, J. Minker, and A. Rajasekar. Extending the semantics of logic programs to disjunctive logic programs. In *Proceedings of the Sixth International Conference on Logic Programming*, 1989.

[62] Y. Lou and Z. M. Ozsoyoglu. LLO: An object-oriented deductive language with methods and method inheritance. In *1991 ACM-SIGMOD Conf. on Management of Data*, pages 198–207, 1991.

[63] M. J. Maher. Equivalences of logic programs. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 388–402, Los Altos, CA, 1988. Morgan Kaufmann.

[64] M. J. Maher and R. Ramakrishnan. Déjà vu in fixpoints of logic programs. In *Proceedings of the North American Conference on Logic Programming*, 1989.

[65] A. Marek and M. Truszczynski. Autoepistemic logic. Technical report, University of Kentucky, 1988. (manuscript).

[66] W. Marek. Stable theories in autoepistemic logic. Technical report, University of Kentucky, 1986. (manuscript).

[67] J. McCarthy. Circumscription – a form of non-monotonic reasoning. *Artificial Intelligence*, 13:27–39, 1980.

[68] J. McCarthy. Applications of circumscription to formalizing common sense knowledge. *Artificial Intelligence*, 28:89–116, 1986.

[69] D. McDermott. A critique of pure reason. *Computational Intelligence*, 1987.

[70] J. Minker. On indefinite databases and the closed world assumption. In *Proc. Sixth Conference on Automated Deduction*, pages 292–308. Springer Verlag, 1982.

[71] R. C. Moore. Semantical considerations on nonmonotonic logic. *Artificial Intelligence*, 25:75–94, 1985.

[72] Y. N. Moschovakis. *Elementary Induction on Abstract Structures.* North-Holland, New York, 1974.

[73] I. S. Mumick and H. Pirahesh. Right-linear and overbound queries. In *Proceedings of the Tenth ACM Symposium on Principles of Database Systems*, 1991.

[74] I. S. Mumick, H. Pirahesh, and R. Ramakrishnan. The magic of duplicates and aggregates. In *Proceedings of the International Conference on Very Large Databases*, 1990.

[75] S. Naqvi and S. Tsur. *A Logical Language for Data and Knowledge Bases.* Computer Science Press, 1989.

[76] J. F. Naughton, R. Ramakrishnan, Y. Sagiv, and J. D. Ullman. Argument reduction by factoring. In *Proceedings of the International Conference on Very Large Databases*, 1989.

[77] J. F. Naughton, R. Ramakrishnan, Y. Sagiv, and J. D. Ullman. Efficient evaluation of right-, left-, and multi-linear rules. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, 1989.

[78] J.-M. Nicolas. Logic for improving integrity checking in relational databases. *Acta Informatica*, 18(3):227–253, 1982.

[79] G. Phipps. Glue: A deductive database programming language. Technical Report TR-CS-90-14, Kansas State University, 1990. Proceedings of the NACLP'90 Workshop on Deductive Databases.

[80] G. Phipps, M. Derr, and K. A. Ross. Glue-Nail: A deductive database system. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, 1991.

[81] H. Przymusinska and T. C. Przymusinski. Weakly perfect model semantics for logic programs. In *Proc. Fifth International Conference and Symposium on Logic Programming*, 1988.

[82] T. C. Przymusinski. On the declarative semantics of deductive databases and logic programs. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 193–216, Los Altos, CA, 1988. Morgan Kaufmann.

[83] T. C. Przymusinski. Every logic program has a natural stratification and an iterated fixed point model. In *ACM Symposium on Principles of Database Systems*, 1989.

[84] T. C. Przymusinski. On constructive negation in logic programming. In *Proceedings, North American Conference on Logic Programming*, 1989.

[85] T. C. Przymusinski. On the declarative and procedural semantics of logic programs. *Journal of Automated Reasoning*, 5:167–205, 1989.

[86] T. C. Przymusinski. Three-valued nonmonotonic formalisms and logic programming. In *Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning*, 1989.

[87] T. C. Przymusinski. Stationary semantics for disjunctive logic programs and deductive databases. In *Proceedings, North American Conference on Logic Programming*, 1990.

[88] R. Ramakrishnan. Magic templates: A spellbinding approach to logic programs. In *Proc. Fifth International Conference and Symposium on Logic Programming*, 1988.

[89] R. Ramakrishnan, P. Bothner, D. Srivastava, and S. Sudarshan. CORAL: A database programming language. Technical Report TR-CS-90-14, Kansas State University, 1990. Proceedings of the NACLP'90 Workshop on Deductive Databases.

[90] R. Ramakrishnan, D. Srivastava, and S. Sudarshan. Rule ordering in bottom-up fixpoint evaluation of logic programs. In *Proceedings of the International Conference on Very Large Databases*, 1990.

[91] R. Reiter. On closed world databases. In Gallaire and Minker, editors, *Logic and Databases*, pages 55–76. Plenum Press, New York, 1978.

[92] R. Reiter. A logic for default reasoning. *Artificial Intelligence*, 13:81–132, 1980.

[93] J. Rohmer, R. Lescoeur, and J. M. Kerisit. The alexander method — a technique for the processing of recursive axioms in deductive databases. *New Generation Computing*, 4(3):273–286, 1986.

[94] K. A. Ross. A procedural semantics for well-founded negation in logic programs. In *Proceedings of the Eighth ACM Symposium on Principles of Database Systems*, 1989. Also Stanford University Technical Report STAN-CS-88-1233. Full version to appear in Journal of Logic Programming.

[95] K. A. Ross. The well-founded semantics for disjunctive logic programs. In *Proceedings of the First International Conference on Deductive and Object-Oriented Databases*, 1989.

[96] K. A. Ross. Modular stratification and magic sets for Datalog programs with negation. In *Proceedings of the Ninth ACM Symposium on Principles of Database Systems*, 1990. Full version submitted for journal publication.

[97] K. A. Ross. Modular acyclicity and tail recursion in logic programs. In *Proceedings of the Tenth ACM Symposium on Principles of Database Systems*, 1991.

[98] K. A. Ross. On negation in HiLog. In *Proceedings of the Tenth ACM Symposium on Principles of Database Systems*, 1991. Full version submitted for journal publication.

[99] K. A. Ross and Y. Sagiv. Monotonic aggregation in deductive databases. (manuscript), August 1991.

[100] K. A. Ross and R. W. Topor. Inferring negative information from disjunctive databases. *Journal of Automated Reasoning*, 4:397–424, 1988. Also appeared as Melbourne University Technical Report 87/1.

[101] Y. Sagiv, 1990. (personal communication).

[102] C. Sakama. Possible model semantics for disjunctive databases. In *Proceedings of the First International Conference on Deductive and Object-Oriented Databases*, 1989.

[103] J. S. Schlipf. Negation by securable failure in logic programming. (manuscript), 1987.

[104] J. S. Schlipf. The expressive powers of the logic programming semantics. In *Proceedings of the Ninth ACM Symposium on Principles of Database Systems*, 1990. (also University of Cincinnati Department of Computer Science Technical Report CIS-TR-90-3).

[105] H. Seki. On the power of alexander templates. In *Proceedings of the Eighth ACM Symposium on Principles of Database Systems*, 1989.

[106] H. Seki and H. Itoh. A query evaluation method for stratified programs under the extended CWA. In *Proc. Fifth International Conference and Symposium on Logic Programming*, 1988.

[107] J. C. Shepherdson. Negation as failure, II. *Journal of Logic Programming*, 2(3):185–202, 1985.

[108] J. C. Shepherdson. Negation in logic programming. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 19–88, Los Altos, CA, 1988. Morgan Kaufmann.

[109] O. Shmueli. Decidability and expressiveness aspects of logic queries. In *Proceedings of the Sixth ACM Symposium on Principles of Database Systems*, 1987.

[110] S. Sudarshan and R. Ramakrishnan. Aggregation and relevance in deductive databases. In *Proceedings of the International Conference on Very Large Databases*, 1991.

[111] M. Takizawa. Distributed database system — JDDBS. *JARECT, Computer Science and Technologies*, 7:262–282, 1983.

[112] J. D. Ullman. Bottom-up beats top-down for datalog. In *Proceedings of the Eighth ACM Symposium on Principles of Database Systems*, 1989.

[113] J. D. Ullman. *Principles of Database and Knowledge Base Systems.* Computer Science Press, Rockville, MD, 1989. (Two volumes).

[114] J. Vaghani, K. Ramamohanarao, D. B. Kemp, Z. Somogyi, and P. J. Stuckey. The Aditi deductive database system. Technical Report TR-CS-90-14, Kansas State University, 1990. Proceedings of the NACLP'90 Workshop on Deductive Databases.

[115] M. H. Van Emden and R. A. Kowalski. The semantics of predicate logic as a programming language. *JACM*, 23(4):733–742, 1976.

[116] A. Van Gelder. Negation as failure using tight derivations for general logic programs. *Journal of Logic Programming*, 6(1):109–133, 1989. Preliminary versions appeared in *Third IEEE Symp. on Logic Programming* (1986), and *Foundations of Deductive Databases and Logic Programming*, J. Minker, ed., Morgan Kaufmann, 1988.

[117] A. Van Gelder. The alternating fixpoint of logic programs with negation. *Journal of Computer and System Sciences*, 1992. (to appear). Available as UCSC-CRL-89-39. Preliminary abstract appeared in Eighth ACM Symposium on Principles of Database Systems, 1989.

[118] A. Van Gelder, K. A. Ross, and J. S. Schlipf. Unfounded sets and well-founded semantics for general logic programs. In *Proc. Seventh ACM Symposium on Principles of Database Systems*, 1988. Full version to appear in J.ACM.

[119] M. Y. Vardi. The complexity of relational query languages. In *14th ACM Symposium on Theory of Computing*, pages 137–145, 1982.

[120] L. Vieille. Recursive axioms in deductive databases: The query-subquery approach. In *Proc. First International Conference on Expert Database Systems*, 1986.

[121] L. Vieille. A database-complete proof procedure based on SLD-resolution. In *Proc. Fourth International Conference on Logic Programming*, 1987.

[122] J. Widom and S. Finkelstein. A syntax and semantics for set-oriented production rules in relational database systems. manuscript, IBM Corporation, 1989.

[123] L. Wong. Inference rules in object oriented programming systems. In *Proceedings of the First International Conference on Deductive and Object-Oriented Databases*, 1989.